# Concurrency Features for the Trellis/Owl Language

J. Eliot B. Moss
Walter H. Kohler

COINS Technical Report 86–40
September 1986

Trellis/Owl is an object oriented programming language being developed as part of the Trellis programming environment by the Object Based Systems group within Corporate Research and Architecture at Digital Equipment Corporation's Hudson, MA facility. Trellis/Owl incorporates support for concurrency. Here we describe the high level language features designed, the rationale for that design, the implementation techniques used, the principles behind them, and experience with the resulting system to date. We show how the object oriented nature of Trellis/Owl influenced the design, and how it affected the implementation.

This report is a reprint of Digitial Equipment Corporation Technical Report DEC-TR-439, August 1986.

## Acknowledgements

The design of concurrency features for Owl was performed by a group of researchers led by Toby Bloom. The group included Craig Schaffert (Digital), Bill Weihl (MIT), and Barbara Liskov (MIT), in addition to the authors. The entire Trellis group reviewed the design. We implemented it, with assistance and guidance from Craig Schaffert and others. All support was provided by Digital Equipment Corporation. The group manager is Ken King.

# Concurrency Features for the Trellis/Owl Language

## Introduction

Trellis/Owl (simply Owl in this paper) is an object oriented programming language being developed as part of the Trellis[1] programming environment. Unlike Smalltalk-80[2], Owl is strongly typed and offers inheritance from multiple supertypes. Owl is targeted to slightly different uses as well: development and maintenance of moderate to large programs. Owl's different objectives strongly affected its design, but actually had little direct impact on the concurrency features or their implementation. For further information on Owl and Trellis see [Schaffert et al. 85, O'Brien 85].

The concurrency features were designed to support moderate to large grain parallelism: an arbitrary operation invocation may be run as a separate activity. No attempt was made to support finer grained concurrency, say at the expression level as for a dataflow architecture, or for vector or array processing. We assumed that the primary source of concurrency would be the logically concurrent tasks requested by the user when interacting with the system. That is, we believed that the applications of most interest to us (programming environments and tools) would not exhibit high concurrency by themselves. This position might have to be reconsidered if the project were one focusing more on concurrency rather than object oriented computing.

We assumed that, in an object oriented environment with large grain parallelism, concurrency is expressed through the interaction of multiple activities on shared objects. This is in sharp contrast to, for example, Hoare's communicating sequential processes [Hoare 85], which organize concurrency around the activities and messages passed between them. Object oriented computing emphasizes modularity of design more in terms of data (objects), with type definitions defining access control and synchronization, rather rather than giving processes the primary role. We feel this approach is more consistent with the overall thrust of Owl, and object-oriented languages in general. However, message passing (e.g., via a user defined message queue data type) is certainly possible within our design, and could be tailored to meet the needs of the application. Since most objects will likely not be shared, we placed the burden of managing access to shared objects on the user. On the other hand, we provided facilities for mutual exclusion and scheduling which allow rather sophisticated concurrency control to be implemented in a modular fashion.

We chose not to incorporate transaction-oriented concurrency in this design. The primary reason for that decision was to allow us to attack and solve an easier problem,

[1]Trellis, VAX, VMS, VAXStation, and MicroVAX are trademarks of Digital Equipment Corporation.
[2]Smalltalk-80 is a trademark of Xerox Corporation.

so as to offer a working base for building the programming environment. We also wanted to see how well a less ambitious design could meet our needs, and to get a better idea of exactly which additional features should be investigated. However, we tried not to include anything in our design that would interfere with a transaction concurrency mechanism. Rather, we believe the features provided will integrate well and assist in implementing transaction support.

Below we review the high level features of the design and further discuss our rationale for choosing them. The design is influenced less by object orientedness *per se* than by data abstraction. There is considerable similarity between the Owl features and ones proposed in [Weihl and Liskov 85] for use with Argus [Liskov and Scheifler 83]. However, there are substantial differences, which we will review later.

We then present the implementation, which has two major aspects. The first is the representation of the specific data types and the algorithms for manipulating them. This aspect is relatively straightforward, so we concentrate mainly on how some novel features are implemented and on how the implementation is tuned to the expected patterns of usage. The second aspect of the implementation is how we achieved appropriate atomicity of execution of various actions in the system. Unlike most interpreter based systems (including the Smalltalk systems of which we are aware), we take a permissive approach to interrupt handling: we allow interrupts and task switches everywhere except in critical sections. Other systems generally check for interrupts in the main interpreter loop, or improve on that in various reasonably simple ways (see, e.g., [Deutsch and Schiffman 84]). We used a number of methods to solve a variety of problems. Other implementers of heap based languages may find our descriptions of these problems and solutions useful.

We conclude with a brief discussion of experience to date with the concurrency features and their implementation.

## Language Features

Our choice of concurrency features for Owl was based on two major assumptions. First, that activities (threads of control) will be used primarily to model conceptually independent tasks. Second, the activity mechanism should be cheap enough that a new activity can be forked whenever a new thread of control would better reflect the task structure of the application. Two implications of the first assumption are that a forking activity may not need to wait for the results of a forked activity, since the two activities are logically independent, and it is unlikely that frequent conflicts over shared data will occur. Activities are not intended to be the principle units of functionality or modularity in Owl; those roles are taken by operations and type definitions (respectively).

The language features we have added to Owl to support concurrency include mechanisms for forking an activity and for awaiting termination of one. We have also provided concurrency control mechanisms to serialize and schedule concurrent access to shared objects by multiple activities.

## The Activity Type

Activities are represented by objects of type `Activity`. New activities are forked by the `create` operation. For example, if `example_act` is a variable of type `Activity`,

```
    example_act := create (Activity, "FOO", {a1, a2, ...});
```

forks a new activity to perform operation `FOO` with arguments `a1`, `a2`, etc. The activity object `example_act` identifies the new thread of control, and provides means for monitoring its status, as well as a handle for debugging. Any operation can be forked as a separate activity.

An activity terminates only by completing the operation it was forked to perform, whether that completion be normal (via a return) or exceptional (via a signal).[3] It cannot be aborted by another activity, not even its parent, since this might leave the objects it was manipulating in an inconsistent state.[4] However, a collection of activities can be designed to coordinate their termination by explicitly setting and checking the state of a shared object.

Generally we do not expect parent activities to want to wait until a child activity completes, but we do think there will be cases when one activity will want to monitor another activity and handle exceptions that arise, or use the results returned by the forked invocation. To support this type of coordination we provide two operations on activity objects: `await` and `await_with_timeout`.

Assume that `A` and `B` are activities. If `A` does `await(B)`, `A` is blocked until activity `B` terminates. If `A` invokes `await_with_timeout(B)`, `A` is blocked until `B` terminates or the wait times out. Timeout is indicated by having the call to `await_with_timeout` signal the exception `timeout` rather than returning normally.

Since an activity may want to start a number of new activities and then wait until some or all of them complete, we also provide an iterator[5] on objects of type `Activity_Set`.

[3] Owl's exception handling is similar to that of CLU (see [Liskov, et al. 1981]) or Ada. Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

[4] The implementation supports more powerful manipulation of activities, intended for use by a debugger utility only.

[5] An *iterator* is an operation that provides members of some collection of objects one by one to a loop body for processing. See also [Liskov, et al. 1977, Liskov, et al. 1981].

An Activity_Set is a set of activities, with some special operations for monitoring completion. We may insert new activities into an Activity_Set, close it (to indicate no more activities will ever be added), and process its members, as they terminate, with the await_next iterator. This iterator finishes only when all activities in the Activity_Set have both terminated and been provided by the iterator for processing, and the set has been closed. Note that Activity_Set has just the operations necessary for handling termination of each member of a dynamically defined set of activities, and no more. It is sufficiently special that it is not really useful to define it as a subtype of Set[Activity].

## Concurrency Control

Since we have assumed that activities interact primarily through the use of shared objects, we provide mechanisms for the implementers of a data type to serialize and schedule operations on shared objects by concurrent activities. The synchronization primitives we provided for Owl are quite basic, but they can be used to implement sophisticated synchronization policies in a modular fashion.

Synchronization in Owl is supported by two complementary mechanisms: *locks* for mutual exclusion, and *wait queues* for waiting and signaling. Locking with waiting and signaling is also the basis for the monitor approach to synchronization [Andrews and Schneider 83], but we did not include the automatic encapsulation provided by monitors, feeling it to be too rigid.

### *Mutual Exclusion Locks*

Our locking mechanism uses a mutual exclusion Lock type with three basic operations: create, acquire and release. In use, a lock object is generally included as a component of a shared object. Acquire and release operations on the lock can be used to guarantee exclusive access to the shared object. Usually, however, these operations would not be invoked directly by the application programmer. Instead, they are used by the system as part of a structured *lock block*, which will be discussed later.

The Lock operations act as follows. To create a new lock object, one uses create. Newly created locks start unlocked. An activity that invokes acquire (via a lock block) is granted the lock immediately if it is currently unlocked. If the lock is held by another activity, the requesting activity is suspended in a first-in, first-out queue associated with the lock. The release operation is used to unlock a lock that is currently held.

Since waiting activities may hold locks while waiting for a new lock, there is potential for deadlock. If lock deadlock occurs, it is detected by the system[6] and one or more waiting activities are chosen as deadlock victims. Each deadlock victim's `acquire` operation will raise the `deadlock` exception. It is up to the programmer to determine how to handle the deadlock, but a deadlock victim should release some locks to enable other activities to obtain the locks and proceed. However, an activity that has raised a deadlock exception is not forced to release its other locks, and it is not prohibited from requesting additional locks. If it requests additional locks, the system will test for deadlock again and this may raise another deadlock exception. After a lock is released, the first activity in the lock's first-in, first-out queue will be allowed to acquire the lock and proceed.

Lock deadlock is not the only potential cause of unbounded waiting. As we will see later, activities can also be waiting for a wakeup signal from another activity that will never occur. The potential for this type of wait/signal infinite wait is a common feature of concurrent programming. No satisfactory solution is known. Consequently, the Owl system does not attempt to detect the situation automatically. A timeout mechanism is provided that can be used to assist in detecting and handling unbounded waits.

### *Lock Blocks*

The *lock block* structure solves the problem of how to unlock automatically at every exit point from a block of code protected by a lock. Lock blocks are similar to critical regions. For example, code that is to be executed only while a lock is held is written within the body of a block of the form:

```
lock l do
    ... protected code ...
end lock;
```

Entry to the lock block will cause the invocation of the `acquire` operation on lock `l` before the body is executed. At the `end lock`, or at any other exit from the scope of the lock block (such as an exception being signaled), `release` is executed on lock `l`. If the `acquire` signals `deadlock`, the body will not be executed and the deadlock exception will be caught by the nearest exception handler enclosing the lock block. Exceptions raised within the body of a lock block are handled in the normal way by exception handlers within the block. Multiple locks can be acquired sequentially by nesting lock blocks.

There is an optional `else` clause allowed in lock blocks. If the `else` is present, an `acquire_nowait` lock operation is invoked rather than `acquire`. The effect of the no wait variant is that if the lock can be acquired immediately without queueing the request, then

[6]Deadlock detection and resolution has been designed but has not yet been implemented.

the normal body of the lock block is executed; otherwise the `else` body is executed. The no wait form of the lock block is:

```
lock l do
     normal body: executed if lock is granted
else
     alternate body: executed if lock is not granted
end lock;
```

While the structured lock block form for setting and releasing locks presented here should help the programmer avoid some common errors, such as forgetting to release a lock, it must be used cautiously. For example, if an unexpected signal arises in a lock block, the lock will be released automatically when the signal is passed outside the block. If the signal arises because the resource protected by the lock is broken, it is the type implementers responsibility to catch the exception and take action to fix or remove the resource.

### *Wait Queues*

Often an activity cannot continue until some condition is satisfied by another cooperating activity. In order for the cooperating activity to establish the condition, it may be necessary for the waiting activity to release temporarily some of the locks it holds so that the other activity can access the required data. Once the cooperating activity has established the necessary state condition, it needs a way to notify the waiting activity. We provide for this type of waiting and signaling by using a data type called a `Wait_Queue`.

The type `Wait_Queue` has four major operations: `create`, `wait`, `wakeup`, and `empty`. A wait queue would commonly be a component of a shared object that is used by cooperating activities. Wait queues allow an activity to:

- indivisibly: release locks, enqueue on the wait queue, and suspend execution; and
- wakeup, reacquire the released locks, dequeue from the wait queue and resume execution.

Let `qid` be a variable of type `Wait_Queue` that has been assigned a newly created wait queue:

```
qid := create (Wait_Queue);
```

The operation `wait(qid)` has the following effect. The invoking activity indivisibly performs the following three actions:

- enqueues itself on the `qid` wait queue;

- releases all locks acquired within the current operation (note that there is exactly one lock per lexcially enclosing lock block); and
- suspends execution.

At a later time, some other activity's execution of `wakeup(qid)` restarts the waiter. However, before the waiter is released from the wait queue, the previously released locks are reacquired. There may be a delay while reacquiring locks; the details of lock reacquisition are discussed below. After lock reacquisition, the waiter continues immediately after the `wait(qid)` statement (with the locks it held before the wait).

The effect of `wakeup(qid)` is to awaken the activity at the front of the first-in, first-out suspended activity list for `qid`. That activity will proceed to reacquire locks, etc. If there are no activities waiting on `qid`, the wakeup is ignored. The wakeup is also ignored if there is already an activity in the process of waking up (i.e., reacquiring locks). In these cases we say the wakeup is *lost*. The rationale for losing wakeups is discussed below.

Since it is useful in implementing some scheduling policies (e.g., readers/writers access to a shared variable), we provide the operation `empty(qid)`, for testing whether there are any waiters on wait queue `qid`. To help detect infinite waits, we provide a timeout mechanism. If the activity that invokes `wait(qid)` has set the per-activity parameter `wait_timeout_interval`, and the activity is not awakened via a `wakeup(qid)` within that amount of time, the wait will terminate with the exception `wait_timeout` rather than returning normally.

In sum, `wait` provides a "hole" in the lock block, and a `Wait_Queue` is similar to a condition variable used with monitors. Our lock block is similar to the `seize` statement of [Weihl and Liskov 85], and our `wait` statement is analogous to their `pause`. However, `pause` results in busy waiting, while we block a waiting activity and explicitly send wakeups later. Busy waiting was perhaps reasonable in their case, since the conditions they are considering are transaction commit and abort, which are relatively asynchronous and independent of the `seize` code. However, we expect lock blocks with wait statements to be used for synchronization and scheduling code: the scheduling state variables are protected by a mutual exclusion lock, and accessed only in lock blocks. In this case blocking and wakeups are reasonable, since we are doing scheduling, not asynchronous notification.

Since we wish to support scheduling of access to shared resources, it is a good idea to help insure that the condition indicated by a wakeup is still true when the awakened activity resumes execution. There are two steps we took towards that goal. First, we give reacquirers priority over normal acquirers of the same lock. This prevents new activities from disrupting the condition established by a wakeup. Second, we discard wakeups presented while an activity is reacquiring locks. If we did not discard such wakeups, the first awakened activity might invalidate the condition expected by the second one. Thus, in our design each activity must send wakeups to the appropriate wait queues. This design reflects our bias towards using the synchronization mechanism for scheduling rather than communication. Also because of this bias, we feel that activities should always block when executing a wait statement. Hence, we discard wakeups sent to empty queues.

While the design decisions just described improve the chances that the condition leading to a wakeup remains true until acted upon, care must still be taken to insure that the condition is not negated between the wakeup and when the lock is released by the activity doing the wakeup. In the design process we considered a number of examples from the literature, and found it easy and natural to achieve this in practice. The next section includes an example of `wait` and `wakeup`.

Since waiting involves releasing and then reacquiring locks, the wait statement raises the issue of deadlock. Giving reacquirers priority over new acquirers helps prevent deadlock, but we went a step further: reacquirers are ordered according to when they are awakened. This ordering prevents deadlock between reacquirers needing the same lock (or the same set of locks). However, deadlocks can occur between reacquirers if they still hold some locks from a containing scope. The reacquire ordering also provide fairness, which is usually preferred in scheduling problems. For a discussion of some alternative concurrency designs, see [Andrews and Schneider 83].

### *An Example*

The fragment of Owl code for `Shared_Char_Buffer` shown below demonstrates how the lock block and wait queue mechanisms can be used to synchronize access to a character buffer shared by producers who insert characters and consumers who remove them. The shared type is constructed from an unsynchronized type `Char_Queue`, by adding a lock component for mutual exclusion, and a wait queue for empty synchronization. A `Char_Queue` is assumed to be an extensible data structure that never overflows (this is readily implemented in Owl).

Note that in the `remove` operation it is not necessary to check for `empty` again after the `wait`. A `wakeup` is done only when the buffer is not empty, and the non-emptiness condition cannot be negated before the first waiter gets the lock and proceeds to remove the newly added item. Furthermore, because reacquirers have priority over acquirers, it is not possible for multiple inserts to occur before a waiting remover gets the first inserted item. This prevents situations where the first waiting remover is successful, but later wakeups are lost so that other waiting removers do not wakeup, even though the buffer is non-empty.

```
type_module Shared_Char_Buffer

component me.mutex    : Lock;
component me.nonempty : Wait_Queue;
component me.buffer   : Char_Queue;

operation create (Mytype) returns (Mytype)
    is allocate
    begin
        me.mutex    := create (Lock);
        me.nonempty := create (Wait_Queue);
        me.buffer   := create (Char_Queue);
    end;

operation insert (me, x: Char)
    is begin
        lock me.mutex do
            insert (me.buffer, x);
            wakeup (me.nonempty);
        end lock;
    end;

operation remove (me) returns (Char)
    is begin
        lock me.mutex do
            if empty (me.buffer) then
                wait (me.nonempty)
            end if;
            return remove (me.buffer);
        end lock;
    end;

end type_module;
```

## Implementation Data Structures and Algorithms

There are four major data types that must be implemented to support the features discussed above: *activities*, *activity sets*, *locks*, and *wait queues*. Implementation of most operations on these objects is obvious. However, the wait statement implementation is more interesting, as is the timeout mechanism, so we describe them after a brief discussion of the data types.

## Activities

For expedience, we implemented activities in two parts: a "permanent", Owl part called an *activity object*, and a "temporary", low level *stack part*.

- An Owl activity object resides in the heap (dynamic storage area). It may be relocated by garbage collection (gc) and exists as long as there are references to it, even if the activity has terminated execution. Activity objects support many of the Owl (i.e., high level) operations on activities, and each contains a reference to its stack part.
- A stack part is similar to a process state block in a multiprogrammed operating system, but is tailored to Owl. It is allocated by requesting space from the operating system, and cannot be moved or reclaimed by gc, though it can be *recycled* and used for another activity later. It consists mainly of the obvious things: a register save area, a subroutine call stack[7], and so on. Stack parts "exist" only until the corresponding activity is terminated.

It is not necessary to implement activities in two, but it is useful because it separates high and low level implementation concerns, and because it allows easier re-use of the bulk of the storage consumed by an activity.

Every stack part has two sets of threading pointers. One thread simply chains together all active stack parts, so the debugger and other tools can find all "live" activities easily. The other thread chains together activities having the same execution state (ready, running, etc.). Here are the execution state lists maintained in our implementation:

- *Ready lists*: There are 32, to support that many priority levels.
- *Running list*: Contains exactly the currently running activity.
- *Lock acquisition list*: One rooted in each lock; holds the activities requesting that lock, in request order.
- *Wait queue list*: One rooted in each wait queue; holds the activities waiting on that wait queue, in wait order.
- *Frozen list*: Contains activities marked by the debugger as currently ineligible for execution (*frozen*), but otherwise ready.
- *Free list*: Contains allocated but currently unused stack parts, for recycling.
- *Reacquire list*: Contains all activities in the process of reacquiring locks after receiving a wakeup in a `wait` statement, in the order in which they were awakened. This list is discussed in more detail later.

In addition, all non-terminated activity stack parts are chained onto a global *active list*, so that the debugger and other tools can find all "live" activities easily.

[7]For simplicity, the call stacks are of a single fixed size, which limits overall recursion depth. This is not as bad as it might seem, since most Owl data is in the heap, not in the stacks.

## Activity Sets

Activity sets are implemented entirely in Owl, using the rest of the concurrency features. The members of a set are stored as elements of an array, and the array is "grown" (replaced with a larger one and the elements copied over) if it fills up. Linear search is used for membership testing, etc. This is deemed adequate for typical activity sets, which are expected to be small. However, it would be very easy to substitute a different implementation for this abstraction in the future.

## Locks

Mutual exclusion locks consist of:

- The current holder of the lock; a null pointer if the lock is free.
- The lock acquisition list: as mentioned previously.
- The earliest awakened activity desiring to reacquire this lock; a null pointer if none.
- A count of the number of activities waiting to reacquire this lock.

The rationale for the last two items is postponed to the discussion of the implementation of the `wait` statement.

## Wait Queues

Wait queues are similar to counting semaphores, but the desired high level semantics requires additional features in the implementation. Recall that Owl wait queues lose wakeups when there is no activity waiting, and when an activity is in the process of waking up (reacquiring locks). For communication with external devices and the operating system, interrupts may be turned into wakeups. However, the Owl wakeup semantics is not always appropriate for these uses. Therefore, the implementation makes losing of wakeups optional. In fact, three separate boolean flags provide a total of eight different wait modes in the implementation, of which perhaps four or five are useful. These flags are:

- Lose wakeups when no activity is waiting: if requested, when an activity is about to wait on a wait queue, any pending wakeups are first discarded, thus guaranteeing that the activity will block. This option is in effect for normal Owl waits.
- Lose wakeups while reacquiring locks: if requested, when an activity is finished reacquiring locks after being awakened, discard any pending wakeups. This option is also in effect for normal Owl waits.

- Wakeup the next activity: if requested, as its last action before continuing after being awakened and reacquiring locks, an activity will perform a wakeup on the wait queue, so as to awaken the next activity in the queue. This is not used in normal Owl waits.

Various combinations of the above options support the normal Owl wait, counting semaphores, events that happen once and thereafter do not cause a wait, and other potentially useful semantics. These features can be made available to the Owl programmer in at least two ways.

- We can add new operations the `Wait_Queue` type, for waiting in different modes; or
- we can provide a separate *type* for each kind of waiting. These types might be presented as subtypes of the Owl `Wait_Queue` type, all of which share the same low level representation, differing only the parameters passed to the implementation when a `wait` operation is invoked at the Owl level.

The first alternative requires the user to insure that each activity that waits on a given queue requests the same wait mode, if consistent, predictable behavior is to be expected. Hence the second alternative is better because it is safer.

To support the above functionality, wait queues consist of: a wakeup count, a first-in, first-out queue of waiting activities, and a reference to the currently waking up activity (a null pointer if there is none).

### The Wait Statement

The Owl `wait` statement requires that locks be granted to reacquiring activities in preference to other requesters, and in wakeup order. To support the preference, each lock has a count of the number of activities needing to reacquire that lock. To support the wakeup order requirement, reacquisitions are granted according to the order of activities in the one, global, reacquire list, which is maintained in wakeup order. To speed lock granting, each lock refers to its next reacquirer. The lock reacquisition count allows us to see if the lock has additional reacquirers. If it does not, then the search for the next highest priority reacquirer can be avoided. We expect that in typical use there will never be more than one reacquirer for a given lock at a time, so we will in practice avoid searching the reacquire list at all. This is a case where we have tuned to expected use, though we support the full semantics as designed.

To support automatic freeing and reacquisition of locks at a `wait` statement, the activity stack part has a separate lock stack. The lock stack has an entry for each lock named in a (dynamically) enclosing `lock` statement (i.e., all locks the activity held just before the `wait` statement). In addition, each entry indicates the Owl stack frame for that lock. Thus `wait` releases and reacquires exactly those locks associated with the current frame (the frame is identified by another slot in the activity stack part). The lock stack also supports debugging, provides consistency checks, and allows automatic deadlock detection. The `lock` statement does lock pushes and pops on the lock stack. At a later date equivalent information may be stored directly in the activity's execution stack, removing the current fixed depth limitation on the lock stack. However, the fixed depth has never been a problem.

### Efficient Timeouts

Since most operations that can timeout probably will not do so in practice, the implementers is challenged to avoid excessive overhead in setting up and removing timeouts. We use the time slice ticks to drive the timeout mechanism, so we avoid extra operating system overhead. However, we still need a data structure that indicates the next activity to timeout. Further, as activities are entered and removed, the data structure has to be maintained efficiently.

We chose to use a balanced binary tree, threaded directly in the activity stack parts, to hold the timeouts. In this tree each node has a sooner timeout than any of its descendant nodes, so the root is the first to time out. Suppose we number the nodes in the tree so that the root is numbered 1, and the children of the node numbered $i$ are numbered $2i$ and $2i + 1$. If we have $n$ nodes in the data structure, the next item is added at position $n + 1$, and then iteratively propagated upwards (by swapping with its parent) until its parent has a sooner timeout (or the next item becomes the root). This arrangement of data and insertion procedure is patterned after the heapsort sorting algorithm. It turns out that upward links are not needed in the tree, since a node's number tells you how to find it from the root. The resulting data structure supports finding the soonest timeout in constant time, and insertion and deletion take $O(\log n)$.

## Additional Implementation Features

A number of features were added to the implementation that were not required in the design for the high level language programmer. The main motivation was to provide convenient support for system functions, a debugger, performance monitor, and other tools. However, some provision was made for possible language extensions (e.g., the addition of a variety of wait modes for wait queues). A number of operations were added to allow the low level state of the objects (activities, locks, and wait queues) as well as the scheduler to be examined. Support for a variety of breakpoints was added. Means for responding to interrupts were provided, including pre-emptive priority scheduling in addition to round robin time slicing of activities having the same priority. Time slicing can be turned off, and its rate adjusted. We have already mentioned that activities can be frozen for debugging. Once frozen they can be examined, and their frames manipulated, though not in as general a way as Smalltalk. Slots for variables, arguments, and temporaries may be examined and updated, and returns or exceptions may be forced. Activities may be identified as being members of zero or more of a moderate number of groups of activities (identified by bit flags in the activity), and such groups may be examined, reorganized, and collectively frozen and thawed.

## Providing Atomicity

Owl takes a permissive approach to interrupt handling and context swapping: interrupts and swaps are allowed except in critical sections. Note that it is not immediately clear exactly what constitutes a critical section. That is, what should be atomic depends on what model and capabilities are offered by the system. We clarify our model and goals below.

Our permissive approach to interrupts and swapping avoids the overhead and additional latency of explicit checks, but requires a more intricate design. We first describe our model and the atomicity problems that must be solved, and then discuss the techniques we used to achieve atomicity.

### Atomicity Problems

The basic requirement is: when an activity is suspended the debugger and gc must be able to act upon it. Both the debugger and gc require the stack to be interpretable. Gc must also be able to find and adjust all pointers, even those in an activity's registers. The requirement that suspended activities be interpretable raises the following problems:

- Frames in the stack must be whole and recognizable. Guaranteeing this requires frame construction and destruction to be atomic.
- An activity stack must not contain uninitialized data, which might confuse the garbage collector or debugger. Hence, local variables must be set to some legal value as frames are built. We chose to use a value recognizable an uninitialized, in order to support detection of uses of uninitialized variables. Conveniently enough, a value of all 0's serves this purpose.
- An activity's saved registers must be interpretable by gc. In particular, gc must be able to distinguish pointers from non-pointers.

The last item has several implications. First, support routines, since they are written in another language where we do not have control over register usage, must be atomic. How we make them so will be discussed in detail below. If we did allow suspension in the middle of a support routine, we likely could not interpret the register contents reliably. The few routines that allow suspension in the middle (e.g., those implementing concurrency control) are specially coded to avoid problems. This is done by copying any Owl pointers that are in local variables to a special place before blocking, and reloading those pointers into local variables upon waking up.

Routines that may suspend in the middle are also vulnerable to being caught there by the debugger. Therefore, we made provision for debugger commands that would break out of the suspension as cleanly as possible. A further consideration is that since most routines are guaranteed to be executed atomically, they do not need to take special care to leave registers in an easy to find location in the stack. Blocking routines do a little extra work to make it possible for gc and the debugger to find registers, and for the debugger to change them effectively (including cutting back the stack, and so forth).

Communication to large external packages where activity switching in the middle may be desirable must be done without giving those packages direct access (pointers) to Owl objects. One approach is to have an array of object pointers and let external packages refer to Owl objects through routines that are given only the index of an object on the table rather than an Owl pointer. To pass information from Owl to external routines, the information is either copied, or Owl objects are created in a non-compacted area (so that they will not move out from under the external package). Reclamation from such an area might be prohibited, or left under user control (an explicit `free` operator). External packages also imply stack areas that do not obey the Owl rules. We have designed and built a simple mechanism for external packages, but it has not yet been fully tested or evaluated.

## Techniques for Achieving Atomicity

Let us briefly review the atomicity requirements that must be met:

- Stack frame creation and destruction, and some other short, frequently occurring, instruction sequences must be atomic.
- Some reasonably short code sequences that are logically inline, but are implemented as assembly language subroutines to save space, must be atomic. (These routines do not follow the usual Owl calling conventions.)
- Most calls to the runtime system must be atomic. Those routines that may suspend in the middle must be specially coded.

- As in any system, there are some critical sections that must execute without the possibility of another activity gaining the processor in the middle.
- Each "global variable" (called a *fixed name* in Owl) has associated initialization code, which is run upon first access to that variable[8]. We must make sure that the initialization code is atomic, to the extent that the initialization is not done twice. (If a second activity requests the value while the first is calculating it, the second is to wait for the first.)

Each of the above problems is solved in a slightly different way. This has partly to do with feasibility of different methods in different situations, but is also related to performance. In particular, building stack frames takes only a few instructions. Adding two or three instructions to the normal case code just to achieve atomicity would be unacceptable. On the other hand, two or three instructions of normal case overhead on a thirty instruction subroutine that is not executed all that frequently may be all right. We describe the methods we used to solve each problem after first giving an overview of the structure of interrupt handling and scheduling in our design.

*Overview of Interrupt Servicing and Scheduling*

Fielding and responding to interrupts in Owl consists of several stages: *registering*, *handling*, *delivering*, and *responding*, each described in detail below. The overall goal is to gain a *response* from an activity. This is done by *delivering* a `wakeup` to an appropriate `Wait_Queue`. *Registering* is the initial recording of interrupt information, and *handling* is the process of deciding when to act on that information.

When the operating system[9] delivers an asynchronous system trap (`AST`), by forcing a call of a previously designated trap handling subroutine, the `AST` handler will call a routine we provide, which will *register* the interrupt. Registration stores an identification of the `Wait_Queue` to `wakeup` and the activity that was running when the interrupt occurred. A few system defined interrupts are delivered to a special `Wait_Queue`, for debugger use; each user defined interrupt can be delivered to any desired `Wait_Queue`. The interrupt information is saved in a ring buffer. Registration acts as a producer, storing into the buffer, and delivery acts as a consumer, reading out of the buffer and posting to `Wait_Queue` objects. Ring buffer access is synchronized with appropriately atomic instruction sequences.

In addition to storing information in the ring buffer, registration must insure that the next step, *handling*, will take place. This is done in different ways depending on the current

[8]Owl requires a fixed name to be initialized some time before the first use. The current implementation executes the initialization code upon the first attempt to use the fixed name.

[9]The implementation we describe runs under `VMS`. However, the techniques and almost all of the code are operating system independent.

status of the system. As will be explained in more detail later, there is a flag indicating whether we are currently in interruptible code. If we are interruptible, then the program counter is saved and the interrupt handler's return point modified so that handling will take place immediately. If we are executing non-interruptible code, we need only set a flag indicating that an interrupt has occurred; the non-interruptible code will check this flag later. If the Owl system was completely idle (pending input/output, for example), then a special signal is made to the operating system, to break out of the idle wait after dismissing the interrupt.

As mentioned previously, interrupts are handled either immediately, or when non-interruptible code is finished. The handling code either immediately calls a routine to *deliver* all pending interrupts, or uses one of the atomicity mechanisms to defer handling to a later time. This is discussed more fully when the mechanisms are described.

*Delivering* interrupts is reasonably simple: each item in the ring buffer is consumed and the appropriate `Wait_Queue` is given a `wakeup`. This may cause some activities to become ready. Interrupt delivery is itself atomic, but when it is finished a new activity may be chosen for execution. The activity awakened by a delivered `wakeup`, when run, will then *respond* to the interrupt. Activity priorities can be adjusted to give response appropriate to the interrupt. This is similar to the technique described in [Goldberg and Robson 83].

Here is a brief summary of the interrupt servicing stages:

- *Registering:* The fact that an interrupt has occurred is recorded.
- *Handling:* The system decides when to act upon the information (immediately, if possible; otherwise at the end of the current critical section).
- *Delivering:* The appropriate `Wait_Queue` is given a `wakeup`, thus notifying the Owl code of the event.
- *Servicing:* The system or application Owl code responds to event as desired.

We now consider the atomicity mechanisms.

### *Very Short Atomic Sequences: Emulation*

The problem in making short instruction sequences atomic is achieving acceptably low overhead. In particular, even if we could protect these sequences with one instruction at each end, the overhead would still be substantial, especially since operation call and return (stack frame creation and destruction) is so common. Our solution to this problem is to have the interrupt handling code detect when we are in one of these sequences and effectively delay handling until the sequence is complete. The sequences are recognized by the occurrence of particular opcodes; the "delay" is achieved by emulating these instructions on the saved state. (Actually, very little of the state is saved, and in many cases the instructions can be directly executed.) Emulation continues until we are out of the sequence, and then handling begins.

Since we are matching based on opcodes (plus the registers and addressing modes used), it is possible that we occasionally emulate instructions unnecessarily. However, because all the sequences are short, we will not waste much time with such unneeded emulation. A more subtle problem is insuring that emulation terminates. We do this by checking for each opcode to be emulated in the same order they occur in the atomic sequences, and after emulating one instruction we continue by performing the *next* check on the next opcode, rather than going back to the start of the emulation tests.

Emulation involves overhead when handling interrupts (we have to check for instructions to emulate whenever we are about to handle interrupts), and is somewhat slower than executing the emulated instruction directly. However, it occurs rarely enough that the cost does not substantially impact overall performance. We chose not to use the single-step trap on the VAX for emulation. Rather, since almost all the registers are intact, once we recognize an instruction requiring emulation, it is usually easy to execute a corresponding instruction (or a few instructions) to update the registers. The program counter has been saved, so branches are emulated by updating the saved value, etc.

Here is an example extracted from our emulation code for the VAX. The sequence to be made atomic is:

```
MOVQ R10,-(SP)
MOVL SP,R10
MOVL #xxxx,R11
JMP  @#yyyy
```

The xxxx and yyyy indicate arbitrary 32 bit values. The code above is part of the entry sequence executed when an Owl operation is invoked. The emulation code is executed in a context where the registers mentioned above have not been changed, so these instruction can be simulated quite directly. During emulation, the saved pc is maintained in R0. Here is the code for emulating the sequence above.

```
...
IF next instruction matches MOVQ R10,-(SP) THEN
    MOVQ  R10, -(SP)          ; emulates instruction
    ADDL2 #3, R0              ; bumps saved pc
IF next instruction matches MOVL SP,R10 THEN
    MOVL  SP, R10             ; emulates instruction
    ADDL2 #3, R0              ; bumps saved pc
IF next instruction matches MOVL #xxxx,R11 THEN
```

```
    MOVL  2(RO), R11           ; emulates instruction
    ADDL2 #7, RO               ; bumps saved pc
IF next instruction matches JMP @#yyyy THEN
    MOVL  2(RO), RO            ; emulates instruction
...
```

*Almost Inline Routines: Location and Restart*

The subroutines that are too long to be inline are grouped into a particular area of memory, and current execution of one of them is detected by examining the saved program counter value to see if it falls in that region. These routines are carefully coded to be restartable, so the handler just backs them up to the point of call (a legitimate location in Owl code), and delivers interrupts at that point.

*Non-Restartable Routines: The Interrupt Flag*

For non-restartable routines, as well as low level critical sections, a flag is set to indicate when we are executing non-interruptible code. This flag is actually a counter, incremented upon entering a critical section, and decremented and tested on exit. Using a counter allows critical sections to be nested, though we have not used this flexibility in the system. A special coding of the interrupt flag simplifies matters here. The second most significant bit of the counter is normally set, and is cleared when an interrupt is registered. The counter value is thus always $\geq 0$, and equals 0 only when an interrupt has been registered and we are not in a critical section. Thus, when the counter is decremented to zero, we have just finished a critical section in which an interrupt occurred.

Decrementing, testing, and conditionally branching to the interrupt handler takes more than one instruction, so emulation is used to insure atomicity of the critical section exit sequence. Emulation could also be used to make the critical section entry code atomic, if it were more than one instruction long.

*Non-Owl Support Code: Return Traps*

This method is a bit more specific to our circumstances, but might still be useful in more general situations. For speed, since it does not require all the features of the VAX calling convention, Owl uses a simplified calling convention. This is part of why the stack frame manipulation is not atomic. Non-Owl code is called using the VAX convention, which modifies a particular register: `fp` – the VAX frame pointer register. By checking to see if the contents of this register is different from the initial `fp` value (saved when an Owl activity is created), we can immediately tell if we are executing Owl code (or almost inline code), or external support code.

If we are in external support code, the handler should defer handling until the external subroutine completes. This is done by tracing up the stack and changing the return address of the frame that would return to Owl. The new return address causes a branch to an entry of the interrupt handler; the old address has been saved in a particular place. We call this technique a *return trap.* Return traps make support routines atomic without any entry/exit sequence. Should an activity block with a pending return trap, the trap is removed, and handling is attempted on the newly scheduled activity (which may result in the placement of a new return trap).

### *High Level Code: Swap Deferral*

To separate low level code, of which there is not much, but which is carefully crafted assembly language and C, from high level Owl code, the interrupt flag mechanism is not used for high level critical sections. This allows us to guarantee, for example, that an activity will not block in the middle of non-interruptible code. It also gives us some bound on the time between interrupt registration and delivery, and thus control over whether or not the interrupt ring buffer might overflow.

For high level critical sections, we provide a *swap deferral flag*, which prevents preemption of the current activity, unless it explicitly blocks itself. One place this is used is in the activity termination code: swaps are deferred while all the activities that are in an `await` on the terminating activity are made ready, one by one.

Swap deferral does not delay interrupt registration, handling, or delivery. However, it does delay actual scheduling of activities awakened by interrupts, and hence holds off interrupt response. Similar to the interrupt flag, the swap deferral flag is a count. However, it is maintained separately for each activity. Thus, if an activity has a non-zero swap defer count, it indicates exactly those places where it is all right to be interrupted by other activities by blocking or explicitly calling a routine that performs a rescheduling.

### *Initialization of Globals: Double Check*

Initialization of globals is made atomic by using an additional slot associated with each variable. This slot gives the identity of the activity initializing the variable. The variable is a candidate for initialization only if its value is a code that means *uninitialized*, namely zero. In that case, a critical section is entered, and a determination is made between the following courses of action:

- The variable has acquired a value between the time we first looked at it and the time we entered the critical section. In this case, just return that value.

- The variable is still uninitialized, but another activity is initializing it. Go to sleep on a special Wait_Queue. When awakened, check the variable's status again.
- The variable is still undefined, and no activity is initializing it. Indicate that the current activity is initializing it, and do so. When done, wakeup all waiters in the special Wait_Queue.

This method causes extra context switches if conflict on initialization is common and the conflict occurs on more than one variable, since a single Wait_Queue is used for all initialization waits. However, we have no reason to expect conflicts to be common.

Of more significance is that we check the value of a variable twice: once outside the critical section, and again inside it. The first check totally avoids the overhead of critical sections and immediately provides the value when the variable is already initialized. The code sequence is short enough that it is done inline. If the variable appears to need initialization, a subroutine is called, which then enters the critical sections and takes action as described above.

## Applicability of Techniques

We feel that the techniques we have described are broadly applicable to language runtime systems that must deal with concurrency.

- Emulation is appropriate for short, stylized code sequences, such as those produced by compiler code generators. It also compensates for any lack of convenient non-interruptible instructions.
- The interrupt flag mechanism is an obvious technique, quite similar to hardware interrupt enable/disable flags. However, it can be used when access to the hardware is impossible, and use of operating system features is too expensive. It works on general sequences of code, but has a few instructions of overhead, in time and space.
- Restartability also bears some similarity to hardware features. It is much less applicable than the previous mechanisms, and requires extreme care in writing the code. Its applicability is further reduced by the difficulty of distinguishing restartable code from non-restartable code, a problem we solved by putting all restartable routines in a special region of memory.
- Return traps are very nice for making subroutines atomic. Their use depends on efficient recognition of the routines that are to be protected that way, on ready interpretation of the stack, and on guarantees that the routines will not bypass the return trap in some fashion. We recognize routines according to calling convention, but location in memory would also work very well. Our routines can bypass the return trap, but only by calling some particular entries, which we fixed to take care of the potential problem.

### Conclusions about Atomicity

The atomicity design and implementation was strongly affected by the presence of garbage collection. It is essential that gc be able to locate all pointers, and to fix the ones to objects moved during compaction. Eliminating compaction does not substantially reduce the problem. In this sense the object oriented nature of Owl had great impact on the implementation. However, features such as inheritance and compile-time type checking did not have much influence on low level implementation; neither did the operating system. We hope that Owl can be ported to other operating systems available for the VAX. Porting to other architectures would require substantially more effort. In our experience this is true of most garbage collected languages: for reasonable performance they require considerable tuning to the specific architecture. However, as we indicated above, the general approach and many of the specific techniques should carry over well.

## Experience

Since the Trellis environment is still under construction and has few users outside the development team, there is little experience with our concurrency mechanism. However, it does appear adequate for building the system, and its collection of tools. Specifically:

- The mechanisms are working dependably.
- They are not too slow. For example, the debugger forks a new activity to process each user command, with no noticeable pause.
- The features are sufficient for a multiprogrammed debugger, for the window system and browser, etc.

The concurrency features are in fact more than just sufficient. Having multiple threads of control available actually simplifies implementation and improves functionality of the system. The debugger is a good example of these effects. Running an interactive debugger as a different activity from the code being debugged has the strong advantage of separating the stacks and thus simplifying the manipulations the debugger must perform on the debugged activity. We can even have multiple debuggers in existence simultaneously, each dedicated to debugging a particular activity. The debugger has also been used effectively in debugging parts of itself. In short, our experience is overwhelmingly positive, though limited.

## Performance

The performance of our implementation is quite good; it is certainly at least adequate for the uses intended. A few preliminary benchmark tests are quoted below. They were performed on a VAX 11/785 and on a VAXStation II (MicroVAX), both running VMS. Each test was run a large number of times, with and without the code to be tested, so that the difference provides the cost of the code being measured. The times given consist of user mode processor time only. None of the operations (except the first activity creation, to allocate a stack part) involves operating system calls. The 785 time is stated first, with the MicroVAX time in parentheses. The numbers are averages of five runs, each with many repetitions of the operation measured, and are accurate to about two decimal places. Most of the support code is written in C; some small parts (e.g., low level task switching) are written in assembly language.

- Task switch: 170 $\mu$sec (260 $\mu$sec).
- Lock acquire/release pair, as in lock block, when the lock is available: 130 $\mu$sec (230 $\mu$sec).
- Synchronized data exchange (via a type similar to `Shared_Char_Buffer`): 0.93 msec (1.9 msec) per exchange (two messages, one in each direction, between a pair of activities).[10]
- Activity creation/destruction overhead: 2.5 msec (4.0 msec) (given stack parts available for recycling). The total cost to create and wind down an activity is somewhat higher, since it includes finding the code to execute given the string name of the operation and the arguments. This additional cost will be reduced by planned changes to string handling and by integration of multiple activities with the compiler.

## References

[Andrews and Schneider 83] Gregory R. Andrews and Fred B. Schneider, "Concepts and Notations for Concurrent Programming", *Computing Surveys*, Vol. 15, No. 1, March 1983, pp. 3-43.

[Deutsch and Schiffman 84] L. Peter Deutsch and Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 System", *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, January 1984, pp. 297-302.

[Goldberg and Robson 83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[10]Note that this includes two acquire/release pairs which block, two waits, two wakeups, and two task switches.

[Hoare 85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[Liskov, et al. 1977] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, Vol. 20, No. 8, August 1977, pp. 564-576.

[Liskov, et al. 1981] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, *CLU Reference Manual*, Springer-Verlag, 1981.

[Liskov and Scheifler 83] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 381-404.

[O'Brien 85] Patrick O'Brien, "Trellis Object-Based Environment: Language Tutorial", Version 1.1, Eastern Research Laboratory, Digital Equipment Corporation, Technical Report 373, November 1985.

[Schaffert et al. 85] Craig Schaffert, Topher Cooper, Carrie Wilpolt, "Trellis Object-Based Environment: Language Reference Manual", Version 1.1, Eastern Research Laboratory, Digital Equipment Corporation, Technical Report 372, November 1985.

[Weihl and Liskov 85] William Weihl and Barbara Liskov, "Implementation of Resilient, Atomic Data Types", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April 1985, pp. 244-269.