Preventing Recursion Deadlock in Concurrent Object-Oriented Systems

Eric A. Brewer * Carl A. Waldspurger
Parallel Software Group, MIT Laboratory for Computer Science

Abstract

This paper presents solutions to the problem of deadlock due to recursion in concurrent object-oriented programming languages. Two language-independent, system-level mechanisms are proposed: a novel technique using multi-ported objects, and a named-threads scheme that borrows from previous work in distributed computing. We compare the solutions, and present an analysis of their relative merits. An expanded version of this paper appears as [4].

1 Introduction

Recursion is a powerful programming technique that allows straightforward expression of many algorithms. Unfortunately, recursion often leads to deadlock in concurrent object-oriented systems. In many systems [2, 14, 11, 5], a method that modifies an object's state cannot even call itself recursively. *Recursion deadlock* occurs whenever an object is blocked pending a result whose computation requires the invocation of additional methods on that same object.

We present two transparent solutions that allow *general* recursion without deadlock. These solutions are transparent in that programs suffering from recursion deadlock will run correctly without change if either solution is incorporated into the underlying system. The first solution is based on *multi-ported objects*, and uses separate communication ports to identify recursive calls. The second solution, *named threads*, draws on previous work in distributed computing, and generates a unique name for each thread in order to detect recursive calls.

A combination of three factors leads to recursion deadlock. First, an object must hold a lock on some state. In systems with at most one active thread per object [2, 14, 11, 5], there is a single implicit lock for the entire object state. Second, the object must make a blocking call to some object (possibly itself), holding the lock while it waits for the response to the call. Finally, the resulting call graph must contain a call back to the locked object that requires access to the locked state, forming a cycle. When these criteria are met, every object in the cycle is waiting for a reply; the objects are deadlocked.

*Supported by an Office of Naval Research Fellowship.
Authors' address: Room 521C, 545 Technology Square, Cambridge, MA 02139. E-mail: {brewer,carl}@lcs.mit.edu.

Aside from simple recursive methods, many patterns of message-passing can lead to recursion deadlock. Programs that manipulate cyclic data structures, use call-backs while responding to exceptional conditions, or implement dynamic sharing mechanisms are all candidates for recursion deadlock. Some programming styles, such as inheritance by delegation, are also prone to recursion deadlock [9].

2 Related Work

A variety of *partial* solutions exist for handling some cases of recursion deadlock. The simplest partial solutions handle only *direct recursion* involving a single object. These amount to releasing the object lock and ensuring that the next method invoked is the recursive call (e.g., by prepending it to incoming message queue), which will reacquire the lock [7].

Another solution for direct recursion is to provide *procedures* in addition to methods [14]. Unlike methods, procedures are stateless and thus do not require locks. In this approach, a method calls a procedure that handles the recursion. This avoids deadlock because there is no lock acquisition for the recursive call. Since procedures only have access to the current object's state, multiple-object recursion is not possible.

A partial solution for *fixed-depth* recursion is the use of selective message-acceptance constructs [14, 2]. For example, ABCL/1 allows calls to be accepted in the body of a method if the object enters a selective "waiting mode". In this case, the recursive call need not acquire the object lock. An explicit waiting mode must be introduced for each level of recursion; if there are too many recursive calls the system will deadlock.

If recursive calling patterns are known in advance, deadlock can be avoided in actor systems by using replacement actors. By cleverly specifying *insensitive actors* that buffer most incoming messages while responding to a few special messages (such as `become`), programmers can write code that explicitly avoids potential deadlocks [1]. Although actor replacement allows deadlock-free code, the complexity of explicitly introducing insensitive actors and behaviors for all possible recursive calling patterns is daunting. In fact, these low-level actor mechanisms (as with *enabled-sets* [13] and *protocols* [3]) could be used to *implement* the solutions we propose without system-level changes.

Techniques for deadlock *detection* [12] also could be applied. Deadlock-detection algorithms examine process and resource interactions to find cycles (assumed to be relatively infrequent), and usually operate autonomously. Although such schemes could be used to detect and recover from deadlock, they would not be practical for fine-grained recursive programs.

3 Computational Model

The model we assume encompasses most contemporary concurrent object-oriented systems. *Objects* encapsulate state with a set of *methods* that can manipulate that state directly. Objects interact only by sending *messages* to invoke methods at other objects.

We divide messages into two categories, *sends* and *calls*. A send is an asynchronous, non-blocking method invocation; it is unidirectional, and has no corresponding reply.

A call is a synchronous, blocking method invocation. After performing a call, the sender waits for a reply. This is analogous to normal procedure call semantics. Any locks held by the sender prior to the call are held until the reply is received. Every call has a matching reply. A set of concurrent calls may also be sent such that each of the calls operates in parallel, and the sender waits for replies from all of the calls before continuing execution.

A *lock* ensures mutual exclusion for some piece of state. A given object may contain several locks. We assume that locks are the underlying primitive synchronization mechanism for mutual exclusion. To avoid complication, we will assume that there is a single implicit lock per object that provides mutual exclusion for the entire object state, as in most contemporary languages [2, 14, 11, 5]. However, the solutions we present can be easily adapted for languages with more sophisticated locking schemes [4].

A *thread* is a single flow of control that performs a sequential computation. A single thread may execute code at several objects. For example, if object A calls object B, the sequential flow of control would execute some code at A, proceed to execute the invoked method at B, and finally continue execution back at A. In a sense, the thread travels with the messages between A and B. This view of threads may differ from that of the underlying implementation. A thread can also *fork* several distinct subthreads by performing concurrent calls. In this case, the original thread is suspended until its subthreads, or *children*, all reply and *join* with the original parent thread.[1]

[1] Joins are relevant only for (blocking) calls.

4 Recursive Call Semantics

In sequential object-oriented systems such as Smalltalk-80 and C++, there is only a single thread of control, so mutual exclusion locks are unnecessary. In these systems, if the call chain generated during a method invocation results in a later invocation on the same object, the recursive call is permitted to modify the object's state. Thus, a recursive call may change the state of an object in sequential object-oriented systems, and there is no deadlock issue.

In concurrent systems, a complete lack of mutual exclusion is not satisfactory. Unfortunately, the addition of locks makes recursion deadlock possible. Ideally, we would like to preserve atomicity while eliminating the potential for recursion deadlock. Our proposed semantics for concurrent systems is consistent with the "expected" behavior in sequential systems. We allow recursive calls to execute without (re)acquiring locks. This change permits descendant threads in the call graph to modify an object's state if a parent thread holds the object's lock.

However, the resulting behavior is undefined if a thread forks subthreads. When a thread forks several children, which thread gets the lock? If all subthreads have access to the object state, they may interfere with one another, exactly the behavior locks are supposed to prevent. The desired behavior is that any of the descendant threads may access the state, but only one at a time. Thus, our proposed semantics is to satisfy two properties: first, descendants must be able to acquire locks held by their ancestors, and second, mutual exclusion must be provided among siblings.

5 Multi-Ported Object Solution

In this section we present a novel solution to the recursion deadlock problem using multiple communication channels, or *ports*, per object. Conventional object-oriented systems assume objects have a single *port* through which all incoming messages arrive. The traditional notion of an object can be relaxed to allow *several* ports. The use of multiple ports to enable different client capabilities, multiple viewpoints, and secure communications is explored in [8]. We demonstrate that the recursion deadlock problem can be solved by providing objects with the ability to create and select ports dynamically.

The recursion problem is solved by creating a new *current port* for each method invocation. This port receives all replies and recursive calls, and persists until the method terminates. An object accepts incoming messages from its current port, and buffers messages addressed to other ports. The current port for an idle object X is the distinguished top-level port P_0^X.

5.1 Handling Messages

Messages that arrive at an object X while it is executing some method H are buffered for later processing. If H is blocked pending the arrival of messages required for further computation, or if H completes, X enters message-reception mode. Messages are accepted if and only if they are addressed to the current port.

Normal object semantics guarantee that for each object, only one method activation exists at a time; objects process messages serially (between state transitions). To permit recursive calls, we weaken this constraint to require that each object maintain a *method activation stack* of pending method activation frames, and guarantee that only the activation at the top of this stack may be actively executing. This corresponds to the stack of procedure call frames found in conventional sequential languages.

The top frame on the method activation stack contains the state of the currently executing computation. Frames other than the top frame contain the state of suspended method invocations that are blocked pending the arrival of reply messages. New frames are pushed on the method activation stack when handling messages other than replies.[2]

Implementation: The procedure for handling a message M accepted by object X is as follows:

If M is a reply, match the reply to its corresponding call. If there are no remaining outstanding concurrent calls, resume the associated blocked thread.

Otherwise, a new frame is allocated on top of the method activation stack. Then a new, locally unique port number P is generated[3] and associated with the frame. This port becomes the new current port; the set of acceptable messages are those addressed to P. Finally, the appropriate method is invoked.

When a method completes, its frame is popped off the method activation stack. The current port is then set to the port associated with the suspended method currently on top of the activation stack.

5.2 Sending Messages

In the following discussion, assume that an object X, during the invocation of its method H in response to message M, is sending message M_Y to object Y. The current port for X during its handling of H is denoted by P_h^X.

M_Y is augmented with a *port binding map*, $\mathcal{B}_{M_Y}$, that associates object names with communication ports. In general, the size of a port binding map $\mathcal{B}_M$ is proportional to the number of distinct objects involved in the processing of message M.

Implementation: The procedure for sending message M_Y to object Y is as follows:

If M_Y is a send, set $\mathcal{B}_{M_Y}$ to `nil`, and then send M_Y to Y at port P_0^Y.

If M_Y is a call, then compute the destination port p and the port binding map $\mathcal{B}_{M_Y}$ to be sent with M_Y. Compute p by searching for the port associated with Y in the port binding map $\mathcal{B}_M$ from message M.
If $Y \notin \mathcal{B}_M$, set p to P_0^Y. Set $\mathcal{B}_{M_Y}$ to be the same as $\mathcal{B}_M$ extended[4] (or changed) to map $X \rightarrow P_h^X$. Finally, send M_Y to Y at port p.

If M_Y is one of several concurrent calls, perform the remaining calls. Otherwise, suspend the current thread pending replies.

6 Named-Threads Solution

The essential elements of named threads are based on *action-ids* from Argus [10], a language for robust distributed computing. The *named-threads* approach to avoiding recursion deadlock assigns each thread a unique identifier that travels with it through every object and message. Every object has a current owner – its currently executing thread, and every message has a name – that of the thread that carries it. In a recursive call, the name of the message matches the name of the owner. This avoids the deadlock that results from attempting to reacquire access to the state.

The simplest cases occur in systems without concurrent calls. Upon acquiring access to the object's state, the thread marks itself as the owner of the object using its name or *thread id*. Upon recursion, the thread id of the message is checked against the thread id of the owner. If the ids match, the incoming message has access to the state. Because the new task can determine that it already has access, it does not wait for the current task to finish, thus avoiding deadlock.

6.1 Concurrent Calls

Most concurrent object-oriented languages allow a single thread to create multiple threads. This wreaks havoc with the simple thread-id solution presented above. As discussed in Section 3, two requirements must be met for the desired semantics: descendants must be able to acquire locks held by their ancestors, and mutual exclusion must be provided among siblings. These requirements lead to the following convention for naming threads.

[2] If the current port is the distinguished top-level port, these messages can be top-level calls or sends. Otherwise, these messages are recursive calls.

[3] Perhaps by simply incrementing a counter.

[4] This extension (or change) need be done at most once per method invocation, not once per call.

Upon creation, a thread is given a unique identifier. When a thread forks subthreads, we extend its thread id for every member of the set of blocking calls; this set is its *call group*. The extension is different for every resulting thread: if thread t performs two concurrent calls, the two new threads are: $t.1$ and $t.2$. If $t.1$ then spawns a two-member call group, there are five threads in total: t, $t.1$, $t.1.1$, $t.1.2$, and $t.2$. Each thread has a unique thread id, and each thread id encodes all of the thread's ancestors. A thread is an ancestor of another exactly when its id is a prefix of the other's id.

6.2 Implementation

The use of named threads requires some extensions to the basic object model. First, an object must have an `owner` field, which contains the id of the current thread. Second, there must be a stack of pending call messages, as explained below, referred to as `ownerStack`. Finally, each message must be marked with its thread id.

Message Acceptance: A message is accepted if and only if it is a descendant of `owner`; that is, `owner` must be a prefix of the message's thread id. The `owner` field initially contains `nil`, which allows all messages to be accepted. Otherwise, an accepted message is either a reply or a recursive call. If the message is a reply, then it is a response to a call spawned by the current owner, and the blocked thread is resumed. (If it did not match the current owner, it would not have been accepted.) If it is a recursive call, the current `owner` is pushed onto the stack `ownerStack`. The thread carried with the message becomes the new owner and executes.

Choosing the Next Thread: Once a thread is started, no new messages are accepted until the thread ends or performs a call. When the thread ends, the previous owner is popped off `ownerStack`. This may make previously unacceptable messages acceptable.

Sending Messages: If a thread performs a call, the object starts accepting messages. Because the thread owns the lock, only descendant messages or the reply will be accepted. A send does not pass on the thread-id, leaving the id field of the message empty. Thus all methods executed in response to a send start new threads.

Concurrent Calls: When concurrent calls are made, the thread id of each member of the call group is extended by a unique number. This guarantees that at most one member of the call group can have access to the object at any given time. The spawning thread retains ownership of the lock until one of the concurrent calls recurses, or until all the replies are received and the method completes.

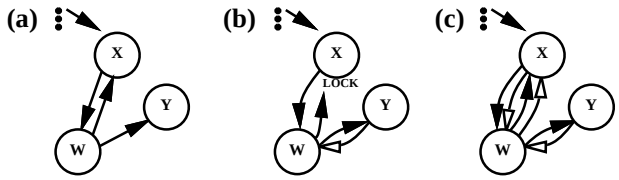


Figure 1: Calling patterns in the abstract query example; call and reply messages are represented by solid and hollow arrowheads respectively.

7 An Illustrative Example

Consider a `do-query` method defined for a `node` object that is connected to other nodes in a graph. When invoked on a node N, this method queries each of N's children concurrently and then returns a function of their replies. The children compute their replies in the same way. As is common in sequential implementations, a node object marks itself visited the first time it is queried, and the `do-query` method returns immediately if an object has already been visited. To illustrate our solutions, we examine this abstract concurrent query algorithm on part of a larger graph. Note that the call tree is isomorphic to the graph; calls are made along the directed edges.

Figure 1(a) presents a graph that involves recursion; the corresponding query in figure 1(b) suffers from recursion deadlock. Since X is blocked waiting for the query to W to complete, the query from W to X never executes. The method at W will not complete until X replies, and X will not reply until the method at W completes. The following two subsections illustrate how the solutions eliminate recursion deadlock in this example.

7.1 A Multi-Ported Object Solution

The following steps, shown in Figure 1(c), trace the message-passing activity for this example. Assume objects W and Y are initially idle, with current ports P_0^W and P_0^Y, respectively. Assume X calls W while its current frame is X_5, with associated port P_5^X.

1. X calls `do-query` at W on port P_0^W, using the port binding map[5] $\mathcal{B} = \{(X \rightarrow P_5^X)\}$.
2. W receives X's call: W starts a new frame W_1 with associated port P_1^W. W concurrently calls X and Y, using $\mathcal{B} = \{(X \rightarrow P_5^X), (W \rightarrow P_1^W)\}$. The call to X is sent to port P_5^X, and the call to Y is sent to port P_0^Y.
3. Concurrently:
   (a) Y receives W's call: Y starts a new frame Y_1 with associated port P_1^Y. Y replies to W and ends frame Y_1.

[5] The port binding map $\mathcal{B}$ would contain additional entries if the current call is part of a larger call chain.

(b) X receives W's call: This is a recursive call involving the path of objects $X \rightarrow \mathrm{W} \rightarrow X$. X accepts the message since it is addressed to X's current port P_5^X. X starts a new frame X_6 with associated port P_6^X. X replies to W, and then ends frame X_6, restoring X_5 as the current frame.

4. W receives the replies from X and Y.
5. W computes the return value as a function of the information returned from X and Y, and sends a reply containing this value to X. W then ends frame W_1, restoring W_0 as the current frame.

7.2 A Named-Threads Solution

This example can be used to illustrate the named-threads technique as well. Objects will be annotated with their current owner: "$X[t.1]$" implies that object X is owned by thread $t.1$.

1. $X[x]$ calls $W[]$, invoking method `do-query`. This leads to $W[x]$.
2. $W[x]$ concurrently calls $X[x]$ and $Y[]$. The message to $X[x]$ has an id of $x.1$, and the message to $Y[]$ has an id of $x.2$.
3. Concurrently:
   (a) $Y[]$ receives the call from $W[x]$. Note that W is owned by x but the calling thread is $x.2$, which implies $Y[x.2]$. The result is calculated and returned to $W[x]$. After the reply, Y is again unowned, that is, $Y[]$.
   (b) $X[x]$ receives the call with id $x.1$ from $W[x]$. This is a recursive call. Since x, the current owner of X, is an ancestor of the calling thread, $x.1$, the call is accepted. The current owner, x, is pushed on to `ownerStack`, and $x.1$ becomes the new owner. $X[x.1]$ replies to $W[x]$, and the previous owner is popped off the stack, which leaves $X[x]$.
4. $W[x]$ receives the replies from X and Y.
5. W computes the return value of `do-query` based on the replies. The result is returned to $X[x]$, and W is again unowned. The ownership is then: $X[x]$, $Y[]$, and $W[]$.

8 Analysis and Comparison

The multi-ported object and named-threads solutions affect the underlying system in three major areas: maintenance of a call stack, overhead for message sending and message acceptance, and an increase in message length.

A send message incurs negligible overhead because there is no call stack, little extra work for handling messages, and no increase in message length.

For the case of blocking calls, each object maintains a stack of pending calls. In the multi-port solution this is the stack of frames, while in the named-threads solution, it is the stack of owners. In general, the size of the pending call stack is unbounded. The height of the stack equals the number of outstanding recursive calls to that object. For single-object recursion, this is the depth of the call chain, while for multiple-object recursion the height will be smaller. The stack is not a side effect of these solutions; it is fundamental to recursion, and corresponds to the procedure stacks used in traditional languages.

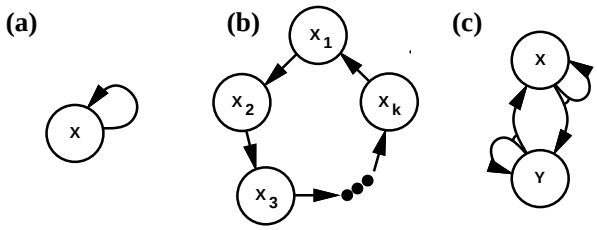
To compare the solutions, it is useful to define two metrics for call chains: *object depth* and *split depth*. The object depth is the number of distinct objects in a call chain. The split depth measures the number of splits in a call chain; an unsplit thread has a split depth of one. Thus if X concurrently calls Y and Z, the split depth at Y is two, one for the original thread and one for the split at X. Using the named-threads notation, the split depth is the number of fields in the id. For example, "$x.1.2$" has a split depth of three.

The solutions differ in the cost of identifying the destination and accepting a message. For multi-ported objects, choosing a destination for a call requires scanning the port binding map to identify the exact destination. The average time for this scan is proportional to the length of the port binding map, which is the same as the object depth of the call chain. Thus the time for locating the destination is proportional to the object depth. For the named-threads solution, the destination is known and the cost is constant.

The cost of message acceptance has a dual behavior. The multi-port solution checks the validity of the port with a single comparison. The named-threads solution must compare the id of the message and the id of the owner. In the worst case, this comparison is proportional to the length of the owner id. Since the length of the owner grows with each split, the cost of the acceptance test is proportional to the split depth. Note that if the object is unowned, the message is a send, or the id starts with a different object, then the test completes immediately.

Message length is also affected differently by the two solutions. For multi-ported objects, the message is extended by the port binding map, which has length proportional to the object depth. For named threads, the message is extended by the calling thread id, which has length proportional to the split depth.

Since the performance of the multi-port solution depends on object depth and that of the named-threads solution depends on split depth, their relative merit



		(a)	(b)	(c)
Message Length	Multi-Ported	O(1)	O(k)	O(2)
	Named Threads	O(1)	O(1)	O(s)
Sending Overhead	Multi-Ported	O(1)	O(k)	O(2)
	Named Threads	O(1)	O(1)	O(1)
Receiving Overhead	Multi-Ported	O(1)	O(1)	O(1)
	Named Threads	O(1)	O(1)	O(s)

Figure 2: Example calling patterns. Arrows denote calls, and arcs joining arrows signify concurrent calls. The table lists the corresponding overheads for each calling pattern.

depends on the call chains encountered in a given system. Figure 2 depicts three possible call chains.

In many applications, call chains involve few objects and few concurrent calls, as in Figure 2(a), and both solutions perform well. Systems with recursion among many objects and few concurrent calls, as in Figure 2(b), would perform better using the named-threads technique. Systems with recursion among small groups of objects and many concurrent calls, as in Figure 2(c), would perform better using the multi-ported object technique.

9 Conclusions

Current object systems place severe limits on the use of recursion, reducing expressive power. The two techniques presented allow fully general recursion in a manner that is transparent to the user. The multi-port solution uses ports to distinguish recursive calls, which places the burden on the sender to identify the correct port. The named-threads solution names each path in the call tree, encoding ancestors in the name. The test for ancestry is used to detect recursive calls, which places the burden on the receiver to identify ancestors. The relative overhead of these solutions depends on application call graphs: those with many objects and relatively few concurrent calls perform better with the named-threads solution, while the multi-port solution performs better in the opposite case.

Recursion is a powerful and important programming technique that causes deadlock in most concurrent object-oriented systems. The solutions presented in this paper provide simple, effective system-level support for general recursion. They can be used by designers and implementors of concurrent object-oriented systems to avoid severe restrictions on the expression of recursion.

Acknowledgements: We thank Barbara Liskov, William Weihl, Adrian Colbrook, Chris Dellarocas, Sanjay Ghemawat, Bob Gruber, Wilson Hsieh, Ken Kahn, Lisa Sardegna, and Paul Wang.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] P. America. "POOL-T — A Parallel Object-Oriented Language." In Yonezawa and Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[3] J. van den Bos and C. Laffra. "PROCOL: A Parallel Object Language with Protocols." *Proceedings of OOPSLA '89*, Oct. 1989.

[4] E.A. Brewer and C.A. Waldspurger. "Preventing Recursion Deadlock in Concurrent Object-Oriented Systems." MIT TR #MIT/LCS/TR-526, Jan. 1992.

[5] A.A. Chien. *Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines.*, PhD thesis, MIT, July 1990.

[6] W.J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

[7] K. Kahn et al. "Vulcan: Logical Concurrent Objects." In Shapiro, ed., *Concurrent Prolog: Collected Papers*, MIT Press, 1987.

[8] K. Kahn. "Objects: A Fresh Look," *Proceedings of ECOOP '89.* Cambridge University Press, July 1989.

[9] H. Lieberman. "Using prototypical objects to implement shared behavior in object-oriented systems," *Proceedings of OOPSLA '86*, Sep. 1986.

[10] B. Liskov. "Implementation of Argus," *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Nov. 1987.

[11] C.R. Manning. *Acore: The Design of a Core Actor Language and its Compiler.* Master's thesis, MIT, Aug. 1987.

[12] M. Singhal. "Deadlock Detection in Distributed Systems," *IEEE Computer*, Nov. 1989.

[13] C. Tomlinson and V. Singh. "Inheritance and Synchronization with Enabled-Sets," *Proceedings of OOPSLA '89*, Oct. 1989.

[14] A. Yonezawa et al. "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1." In Yonezawa and Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.