Destructors, Finalizers, and Synchronization

Hans-J. Boehm
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
Hans_Boehm@hp.com

ABSTRACT

We compare two different facilities for running cleanup actions for objects that are about to reach the end of their life.

Destructors, such as we find in C++, are invoked synchronously when an object goes out of scope. They make it easier to implement cleanup actions for objects of well-known lifetime, especially in the presence of exceptions.

Languages like Java[8], Modula-3[12], and C#[6] provide a different kind of "finalization" facility: Cleanup methods may be run when the garbage collector discovers a heap object to be otherwise inaccessible. Unlike C++ destructors, such methods run in a separate thread at some much less well-defined time.

We argue that these are fundamentally different, and potentially complementary, language facilities. We also try to resolve some common misunderstandings about finalization in the process. In particular:

- The asynchronous nature of finalizers is not just an accident of implementation or a shortcoming of tracing collectors; it is necessary for correctness of client code, fundamentally affects how finalizers must be written, and how finalization facilities should be presented to the user.

- An object may legitimately be finalized while one of its methods are still running. This should and can be addressed by the language specification amd client code.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage Collection*; D.4.1 [**Operating Systems**]: Process Management—*Threads*

POPL'03 January 15–17, 2003, New Orleans, Louisiana, USA

General Terms

Languages, design

Keywords

Deadlock, destructor, finalization, garbage collection, synchronization, thread

1. INTRODUCTION

The following sections describe two mechanism for associating cleanup code with objects: C++ destructors and Java/Modula-3/C# finalizers.

C++ destructors are cleanup actions executed primarily when a program variable goes out of scope. The prevailing opinion is that C++ destructors are clearly useful. And there is some agreement on how they should be used (cf. "Resource Management" in [14]).

Many programming languages that provide automatic, garbage-collected memory management also provide *finalizers* to clean up heap objects sometime before collection. Unlike destructors, finalizers are often viewed as "... unpredictable, often dangerous, and generally unnecessary ... can cause erratic behavior, poor performance, and portability problems" ([1], taken slightly out of context, in section entitled "Avoid finalizers").

A common view is that the asynchronous nature of finalization is a consequence of the use of a tracing garbage collector which detects garbage only at periodic intervals, and that it might become more useful if traditional reference counting were used as much as possible, so that most finalizers could be run synchronously at the precise program point at which an object became inaccessible.[1] It has also occasionally been claimed that if finalization must be used, one should avoid locking in finalizers. We point out that both claims are wrong, though the latter is somewhat motivated by common language implementation bugs.

We argue that although finalization is rarely needed, it is often critical when it is needed. In these cases, its absence would render the garbage collector useless. Furthermore, although finalization timing is certainly indeterminate, it should not lead to unreliable programs.

Some of the observations underlying this paper have been previously made by others. We attempt to point out such prior work. But we are not aware of any published work that thoroughly discusses the implications of those observations, either on the writing of correct client code, or on

[1]This has been repeatedly claimed in various C++ discussion groups, and less frequently on **gclist@iecc.com**.

the underlying design of the finalization facility, and particularly on the futility of designing systems for synchronous finalization.[2]

Although our observations should perhaps be obvious, there is no shortage of empirical evidence that they are not. Nearly every new Java implementation seems to initially execute finalizers synchronously from the garbage-collecting thread, even though this is prohibited by the language specification. The current C# specification[6] appears to allow finalizers to be run synchronously[3], in spite of the fact that, as we argue below, this makes it largely impossible to write deadlock-free finalizers. A number of common memory management implementations for C++ (*e.g.* Boost `shared_ptr`[4]) implement essentially synchronous finalization.

The issues are certainly no better understood among the authors of client code than among language implementors. A large fraction of the finalization uses that we have seen involved unintended races due to insufficient synchronization in the finalizer.

To make this discussion more concrete, we will assume a multi-threaded environment. As we will see below, finalizers (like Unix signals or hardware interrupts) essentially introduce their own thread of control. Thus various issues become easier to express if threads are already present. None of our issues disappear in a single-threaded environment, but they must be addressed with different, and probably less natural, techniques[7].

Although our approach to finalization will deviate slightly from Java semantics in ways we discuss below, we will use mostly Java terminology throughout this paper. In particular, a method or an object is *synchronized* if it acquires a lock.

It is worth remembering throughout all of this that cleanup actions, like memory management issues in general, really become of interest only for at least medium-sized programs manipulating relatively complex data structure. And they become far more interesting for systems that try to rely on modularity so that they do not have to rely on a single implementor understanding the entire system. Unfortunately none of these lend themselves to inclusion as examples in a paper. Thus our examples will tend to be somewhat incomplete.

2. C++ DESTRUCTORS

The C++ language allows objects to have an associated destructor method.[5] When a stack allocated object goes out of scope or is explicitly deallocated using `delete`, the object's destructor is automatically called. This mechanism can be viewed as largely a syntactic convenience, but it does have several advantages over explicit calls to a "destroy" method:

1. Class inheritance is handled correctly by also invoking superclass destructors.
2. It can be guaranteed that any stack allocated object will be destroyed exactly when the object goes out of scope. There is no opportunity to forget to make the call.
3. As a particularly important instance of the last point, destructors significantly simplify cleanup of stack allocated objects in the presence of exceptions.

Since it is easier to handle object cleanup this way in C++, it is common to try to move any kind of resource acquisition into a constructor, and the corresponding deallocation into the destructor. This approach is commonly referred to as "resource acquisition is initialization" or RAII[14]. [6]

As a particularly illustrative and common example, destructors are often used to release locks.[7]

Thus

```
{
    L.lock();
    f();
    L.unlock();
}
```

would be replaced by

```
{
    scoped_lock sl(L);
    f();
}
```

where the constructor for *scoped_lock* acquires the argument lock and saves its identity, so that the destructor can release it. If `f()` raises an exception, the destructor is invoked during stack unwinding, so that L is still released. Thus it is guaranteed that L will be released at block exit.[8]

Destructors are guaranteed to be executed at a well-defined program point. This property makes them useful for cleanup actions that have immediately visible side-effects, such as releasing locks or closing windows.

Destructors are also executed when heap objects are explicitly deallocated, using `delete`. As we will argue below, this works well if the programmer is aware of exactly which objects are deallocated at what point. However, it becomes highly problematic if some automatic or semi-automatic mechanism is used to perform deallocations implicitly.

[2]Some of this paper is an expansion of the web pages we previously made available at **http://www.hpl.hp.com/personal/Hans_Boehm/gc/det_destr.html** and **http://www.hpl.hp.com/personal/Hans_Boehm/gc/finalization.html**

[3]To our knowledge, C# implementations generally do the right thing, and use a separate finalization thread. One of them temporarily did not. See **http://bugzilla.ximian.com/show_bug.cgi?id=31333**.

[4]See **http://www.boost.org/libs/smart_ptr/shared_ptr.htm**

[5]Note that the term "destructor" in the C# language definition corresponds to our usage of the term "finalizer".

[6]Although this is designed to apply to all objects, it seems to be more useful for stack allocated objects than for statically allocated ones. The author knows of more than one large project that has abandoned use of this facility for statically allocated variables due to the difficulty of ensuring that constructor invocations occur in a safe order. Its applicability to heap objects is discussed below.

[7]See for example **http://www.boost.org/libs/thread/doc/mutex.html**.

[8]In our opinion, this raises a stylistic issue, in that there may be no way to see whether a lock is being released at the end of a block without reading the entire block. Thus we assert only that it's common practice, and leave the reader to judge whether it's desirable.

3. FINALIZERS

Xerox PARC Cedar[13][9], Modula-3[10], and our garbage collector for C/C++[2], among others, allow a finalization function to be associated with an object. The finalization function is executed sometime after an object can no longer be accessed except by the finalization method for the object itself.[10] It is given access to the object itself. Finalizer invocation may be arbitrarily delayed in unspecified ways.

Unlike destructors, finalizers are attached to heap allocated objects, typically in contexts in which object deallocation is implicit. They are not used with stack allocated objects, or objects whose lifetime always ends at a syntactically determinable program point.[11]

We will refer to objects with an associated nonempty finalization action as *finalization-enabled*. Finalization-enabled objects which are not reachable from ordinary roots will be referred to as *finalization-ready*.[12]

Typically finalizers are implemented by keeping pointers to all objects with nonempty finalizers in a separate set data structure F. During initial tracing, the garbage collector ignores this data structure. The garbage collector then also traces (i.e. marks or copies) objects reachable by following chains of *one* or more pointers from the objects in F. Any objects that are in F but were not traced during either phase become eligible for finalization. Finally F and the objects it references directly are traced to ensure that objects needed by finalizers are retained by the garbage collector.

The finalizer for an object p may assign p to, for example, a static class member. In that case p may remain live indefinitely after its finalizer has run. This is sometimes, perhaps incorrectly, referred to as *resurrection*. It differs from destructor behavior, but causes no real problems.[13]

3.1 Alternatives

PARCPlace Smalltalk (see [9] for a discussion) provides a slightly different finalization model. An object can be referenced by a *weak array*. Such references are initially ignored by the garbage collector when determining object reachability, but cleared once an object is collected. When this happens, the weak array object is notified that one of its entries has disappeared.

This differs from what we have been discussing in two ways:

- Finalization is combined with a "weak pointer" facility. (Weak pointers are pointers that do not prevent a garbage collector from viewing an object as inaccessible.) The same combination is used in later versions of the Cedar facility[14] or in Java's `java.lang.ref`.
- More interestingly, a *different* object is notified of the unreachability of an object.

Effectively, the notification message can be viewed as a finalizer invocation, but with the difference that the finalizer function does not have access to the unreachable object itself, and hence that object can be immediately reclaimed.

Others have also argued in favor of notifying another object[15]. The approach has its advantages in terms of programming style, and may well make finalizers more comprehensible, since "object resurrection" can no longer occur. A priori it has some advantages when it comes to ordering issues. (See appendix A). However, it can be emulated in our model, at least at this level of detail, by adding a possibly empty "executor" object p', a pointer from p to p', and registering a finalizer only for p'. Thus it doesn't really affect our discussion, and we do not consider this alternative further here.

Java finalizers differ from Modula-3 finalizers in that objects may be finalized even if they are reachable from *other* finalization-enabled objects. This affects the implementation only slightly. The impact on client code is discussed in appendix A.

C# "destructors" are really finalizers in our terminology, and behave essentially like Java finalizers.

3.2 Example uses of finalization

The addition of a finalizer-like mechanism is well motivated in [5]. Here we look at a few additional examples to motivate and clarify our discussion. The third one will be used to discuss some of the synchronization issues which are central to our discussion.

In general, finalizers should be used to reclaim resources, other than garbage collected memory, when timing of the resource reclamation is not critical. In the case of scarce resources, explicit deallocation is usually preferable. But as we see in the second example below, it is not always practical.

3.2.1 *Legacy libraries*

Perhaps the most common use of finalization is to accommodate libraries written for `malloc/free` memory management in a garbage collected environment. Typically such libraries provide an explicit deallocation function d that must be invoked when an object l managed by the library is no longer needed. If the object l is used by a garbage collected object g, then it is common to invoke d from g's finalizer. This may unnecessarily delay reclamation of l, but this is typically no more of a problem than the delayed deallocation of memory inherent in a tracing garbage collector.

3.2.2 *Files in ropes*

Occasionally complex data structures contain embedded non-memory resources. A good example of that is the "rope" data structure described in [3].[16]

[9] Rovner[13] describes an early version of Cedar finalization. Later versions are closer to the Modula-3 version.

[10] To simplify matters, we ignore special "weak" references that are ignored in making this determination. Such a facility is often combined with a finalization facility, and the combination can be quite useful.

[11] Ada's use of the term "finalization" corresponds to our "destruction".

[12] We avoid the term *finalizable* since it has been used to mean either of these. See for example [9] and [8].

[13] Nonetheless it is often viewed as problematic. The only reason we can identify is that at least in the original Java specification, there was no way to reenable finalization to be run again when the p becomes inaccessible (again). This restriction appears to be fairly arbitrary and Java-specific. It is not shared by, for example, Modula-3 or Xerox Cedar.

[14] Early versions used "package refs". Effectively the garbage collector ignored a predetermined number of references to finalization-enabled objects.[13]

[15] See for example, the discussion of "death notices" on **gclist@iecc.com**.

[16] A similar, but explicitly reference counted, data

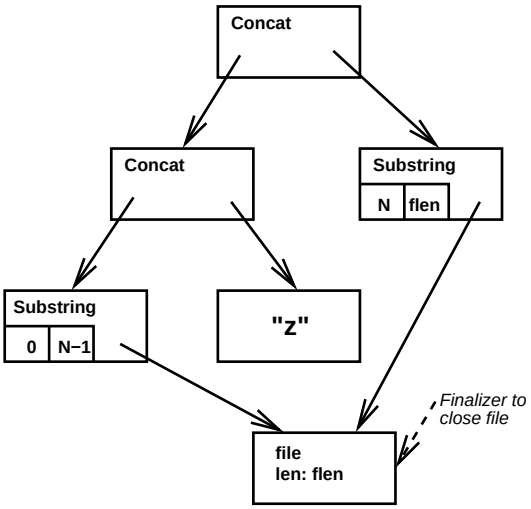

Figure 1: Slightly edited file as a rope

Here a string is represented as a binary tree or DAG. Each leaf is a string constant. Interior nodes represent the concatenation of the strings represented by the left and right subtrees. This representation allows constant time concatenation of arbitrarily long strings. If some care is taken to keep leaves small and trees balanced, it also allows efficient substring operations on long strings. These make it possible, for example, to build an efficient text editor which represents the entire file being edited as a single rope.

A common extension is to allow file descriptors instead of explicitly stored strings as leaves, and to allow interior nodes representing unevaluated substring expressions. These allow an editor to operate on a large file represented as a rope, without reading the entire file into memory. After a single character insertion of the letter "z" after the Nth position in a large file, the in-memory representation of the resulting file might be as in figure 1.

In general it is difficult to predict when the file nodes in such a data structure will be dropped. For example, in the Cedar programming environment[13] ropes are used as the standard string representation, and are thus themselves embedded in many other data structures. Even in the editor case, an embedded file descriptor will be dropped if all characters in the original file are replaced. Thus if we wanted to explicitly close a file, we would have to be able to determine when a file node is no longer accessible from any accessible ropes; effectively we would have to redo exactly the work already performed by the collector.

Finalizers are well-suited to handling this problem. We simply attach a finalizer to each leaf containing a file descriptor. Sometime after that leaf becomes unreachable, the finalizer closes the file. This may lead to files being kept open longer than necessary; but for typical applications such as the text editor example, very few files tend to be embedded in ropes in this manner, so this is unlikely to be an issue. As we will see below, we can potentially reclaim file descriptors more promptly with some help from the client code and file open routines.

structure is included in the SGI and GNU versions of the C++ standard template library. See **http://www.sgi.com/tech/stl/Rope.html**.

3.2.3 External object data

Assume that we have a class C, such that all instances of C maintain at least some of their state (a C_impl instance) in a permanently reachable array. This allows us to easily share a single C_impl between multiple Cs containing the same data. The global array allows us to easily search for an existing C_impl to reuse. We'll see another reason to do this in the next example.

This looks something like:

```
class C_impl {
  // Some stuff needed by C instances.
  T data;
  public C_impl(T d) {data = d;}
}

class C
{
  // The following two arrays are protected
  // by the lock on the impls array.
  static C_impl impls[] = new C_impl[N];
  // The number of Cs sharing the
  // corresponding C_impl.
  static int impl_use_count[] = new int[N];

  int my_index;  // impls index for my rep.

  public C(T t) {
    synchronized(impls) {
      for (int i = 0; i < N; ++i) {
        if (impl_use_count[i] > 0 &&
            <impls[i] reusable>) {
          my_index = i;
          return;
        }
      }
      my_index = first_available();
      impls[my_index] = new C_impl(t);
      impl_use_count[my_index] = 1;
    }
  }

  protected void finalize() {
    synchronized(impls) {
      --impl_use_count[my_index];
      if (impl_use_count[my_index] == 0) {
        impls[my_index] = null;
      }
    }
  }

  static int first_available() {
   // Caller must hold impls lock.
   for (int i = 0; i < N; ++i) {
     if (impl_use_count[i] == 0) return i;
   }
   throw ...
  }

}
```

The data structure is pictured in figure 2. Effectively C

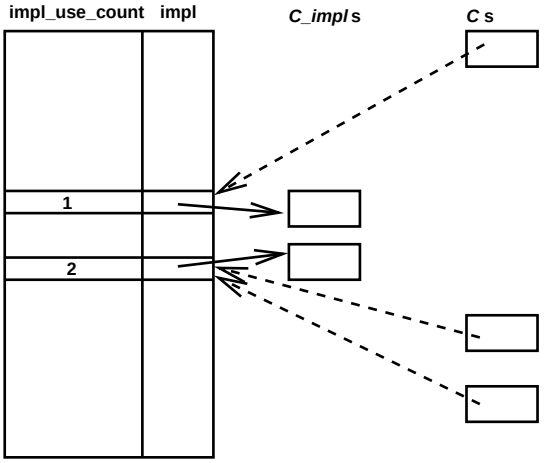

Figure 2: Data structure for example 3.2.3

objects contain only an index *my_index* into a global table *impls*. The real instance data is referenced by the table *impls*. When a C is no longer otherwise reachable, its finalizer is invoked. This updates *impl_use_count* and possibly explicitly removes a *C_impl* from the table so that it can be reclaimed.

If we had put the *C_impl* state directly into C objects instead, we would have needed an auxiliary table to find candidates for reuse. This table would have prevented the garbage collection of C objects without a mechanism such as weak pointers.

Note that the finalizer here must acquire the class lock since it updates shared data structures.

3.2.4 Removal of temporary files

Occasionally it is necessary to provide for reliable cleanup of certain resources before process exit. Removal of temporary files is often an example of this.

As is discussed in appendix A, finalizers by themselves cannot provide this facility since the finalization mechanism doesn't have the necessary ordering information.

Here we outline how to use finalizers to remove temporary files as they are dropped during program execution, while making it possible for an explicit routine to remove those that are remaining at process exit.

Since finalizers cannot guarantee to remove all such files, we will need an explicit routine *cleaner* to be called before process exit. It will be the clients responsibility (as it must be) to ensure that this is only called after the last use of a temporary file, but before the removal of any other objects needed by *cleaner*.

This *cleaner* routine will need access to the system resources which are still allocated. One way to accomplish this is analogous to our previous example. We keep an explicit, always reachable, table T of temporary files.

Clients are not given direct access to T. These clients instead access finalization-enabled temporary file objects containing primarily an index of, or a reference to, the corresponding file information in T. The data structure is outlined in figure 3.[17] Finalizing a temporary file object causes removal of the file, and the corresponding entry from T.

[17]Note that references to the client-visible object are synchronized, i.e. acquire the lock on the object. This is necessary for reasons discussed below. Accesses to T must also be synchronized, since it must support concurrent access.

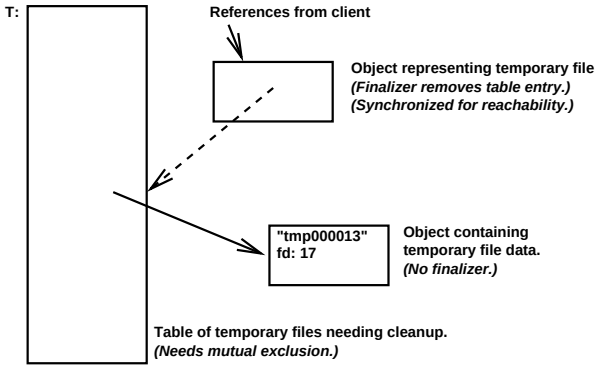

Figure 3: Guaranteed cleanup of temporary files

Cleaner removes any entries left in T at process termination.

3.3 Locking of finalizers

Many finalizers should guard against concurrent access to the underlying object by acquiring a lock. There are two reasons for this:

1. There is usually not much of a point in writing a finalizer that touches only the object being finalized, since such object updates wouldn't normally be observable. Thus useful finalizers must touch global shared state (e.g. static fields of Java classes). Generally this shared state can be concurrently updated by client code outside finalizers. In a language such as Java, multiple finalizers for the same class may also run concurrently in separate threads (even if the original application is single-threaded!), so that we must ensure mutual exclusion between finalizers.

   Sometimes, as in our ropes example above, the only shared state that is touched is inside the operating systems kernel or low level system libraries, so that the necessary synchronization is hidden. In many other cases, *e.g.* in our external object data examples, user-visible locks are needed.

2. Object x may be finalized before $x.foo()$ has finished executing. This is possible since $x.foo()$ may no longer need access to x past a certain point in the method, and thus there is no reason for x to be treated as reachable by the garbage collector past that point. Consider specifically:

```
class X {
    Y mine;

    public X() {
        mine = new Y();
        ...
    }
    public foo() {
        ...;
        mine.bar();
    }
```

```
    public void finalize() {
        mine.baz();
    }
}
```

where *mine* is a subobject not visible to the outside world, and whose methods can thus not be called except through the corresponding X object. Assume that only one thread uses object x of class X, and that the last action on x is to call $x.foo()$. When foo calls *mine.bar*, x may no longer be reachable. It is quite possible that, at this point, the register holding x will have been reused, and the stack frame for foo no longer exists due to tail call optimization. Thus the finalize method may be invoked at this point, and `mine.bar()` may run concurrently with `mine.baz()`. If both update the same data structure, this is nearly certain to be unacceptable.

We can ensure that `mine.bar()` and `mine.baz()` run sequentially by adding locking, *e.g.* by marking foo and $finalize$ as synchronized.[18]

This is at odds with the occasional advice to avoid synchronization in finalizers.

3.4 Reachability and Optimization

Most specifications for languages supporting finalization are intentionally somewhat vague about finalizer execution timing in both directions. They not only allow finalization to be delayed, but also allow finalizers to be executed earlier than might be expected, as the result of compiler optimizations. We first concentrate on the latter.

We saw in the preceding section why this might happen with private subobjects. However, things can get even less pleasant.

Consider an example resembling the external object data example from section 3.2.3. In this case, assume that we have objects of class D containing an index *my_index* to the external array A. Each entry in A is used by at most one D object. It contains either a `null` pointer, if it's not currently used by any object, or otherwise a reference to an object containing a *counter* field. The finalizer for D resets the array entry to `null`. Assume that D has a method foo that simply counts the number of its invocations by incrementing the corresponding *counter*:

```
void foo() {
    ++A[my_index].counter;
}
```

Now consider what happens when an instance d of D is accessible only by a single thread, and it is last accessed in the loop

```
    for (int i = 0; i < 1000000; ++i) {
        d.foo();
    }
```

[18] Interestingly, as we discuss in the next section, it would probably suffice to declare only foo as synchronized. That would ensure that x remains live until the release of the associated lock, and thus sufficiently delay finalization to ensure that the methods are run sequentially. As discussed briefly in [9], other methods for guaranteeing the liveness of x may be thwarted by compiler optimizations.

A compiler might inline foo, observe that $d.my\_index$ is loop invariant, and thus simply keep it in a register. It may then observe that d is now dead, and no longer needs to be kept anywhere. If the compiler eliminates d in this manner, and the collector then runs while the loop is executing, it will discover that d is unreachable, and should thus be finalized. The finalizer will set $d.my\_index$ to `null`, and cause the next increment to fail.

The apparent conclusion from this is that we need to be a bit more precise about what references can be eliminated by the compiler, and how soon finalizers can be run. Unfortunately, we are not aware of any language specifications that are sufficiently precise about this.

As an example, the Java Language Specification[8] specifies that "A reachable object is any object that can be accessed in any potential continuing computation from any live thread. Optimizing transformations can be designed that reduce the number of objects that are reachable to be less than those which would naively be considered reachable." We believe that in light of the above example, this is not sufficiently precise.

A minimal solution to this problem for Java would be to insist that an object is reachable at least until the lock on the object has been released for the last time. This would eliminate the problem in the above example once we make foo synchronized. It could also be used to eliminate corresponding problems in examples 3.2.3 and 3.2.4.

Some of the constructions in appendix A of this paper, as well as, for example, "Finalizer Guardian" construction in [1], require that certain other potentially unaccessed objects need to be considered reachable. In particular, if there is a reference from A to B, and A is reachable, we need to know that B will not be finalized. We will continue to make this assumption were necessary, since existing code relies on it and, to our knowledge, all existing implementations satisfy it. (It would be acceptable to restrict this assumption to `final` and `volatile` references.)

The situation in C# appears to be very similar, and in need of similar solutions. For some other languages, the definition of "reachability" is even less precise.

3.5 Finalization must be asynchronous.

Java requires finalizers to run inside a thread holding no user-visible locks. Those virtual machines that implement this correctly appear to universally do so by running finalizers in a separate thread or threads, once the collector has determined, at its leisure, that an object is eligible for finalization.

To understand the reasoning behind this requirement, it is useful to consider, instead of Java, the case in which locks cannot be reacquired by a thread. (For example, pthread locks[11] work this way by default.) If finalizers can be run at any point in any thread, a finalizer requiring lock L may coincidentally be run in a thread that already holds lock L. The attempt to reacquire the lock results in deadlock.

If a finalizer needs to acquire a lock, there is very little the programmer can do to avoid this scenario. Even if the programming language, like Java, allows locks to be reacquired by the same thread, the situation does not improve. Instead of an obvious failure, we will allow two logically separate operations, namely the client hread holding L and the finalizer, to hold L at the same time. The example in appendix B illustrates that this can result in intermittent incorrect

execution.

This issue was apparently known to the authors of [8]. Unfortunately, most major Java implementations appear to have implemented this incorrectly in their initial versions.[19]

We argue that this observation is equally applicable to environments that traditionally use reference-count based garbage collection, and even if finalization is implemented based on destructors.

For example, it has often been argued in the context of automatic memory management for C++, that destructors for heap objects should behave just like destructors for stack objects, and that they must be run the instant the last reference to an object is dropped, *e.g.* in response to a reference counter decrement. And this is exactly what is commonly done by C++ reference count implementations (cf. [4]). Python[16] uses a similar approach. In both cases destructors[20] are used to implement finalizers.

But this encounters exactly the same problems that were encountered by early Java implementations: Due to the presence of reference counting, which is designed to free the programmer from worry about deallocation timing, the timing of finalizer execution is no longer transparent. Finalizers may run in response to a reference decrement at any assignment or block exit, and hence appear to the programmer to be asynchronous. Deadlocks involving a lock held by the thread processing the reference count decrement and required by the finalizer are neither predictable nor avoidable.

It is impractical for a tracing garbage-collector to run finalizers at exactly the point in thread execution when an object becomes finalization-ready. But even if it could do so, this would in fact make it much harder, rather than easier, to write correct code.

Appendix B expands on the "external object data" example from section 3.2.3 to give a concrete illustration of how synchronous finalization in response to a reference count decrement can fail unexpectedly. The example also illustrates that module abstraction boundaries effectively make it impossible to avoid such disasters, even with "deterministic" reference-counting garbage collection.

The often repeated complaints that "finalizers are unpredictable" (cf. [1]) is a necessary feature, not a deficiency.

3.5.1 Explicit Finalizer Invocation

Java provides a method `System.runFinalization()` which explicitly forces finalizer invocation. Various finalizer implementations and proposals (cf. Cedar [13], Guardians[5], the Ellis-Detlefs safe C++ proposal[7] or `java.lang.ref` in Java2) do not always run finalization procedures implicitly, but may instead simply enqueue finalization-ready objects, leaving it to the client to read the queue and invoke the appropriate procedures.

These allow finalizers to be used safely in single-threaded environments, by requiring the client to explicitly invoke finalizers when it is safe to do so, i.e. outside code sections that should be executed atomically. This appears to be the safest way to handle finalization in single-threaded code, although it has some clear disadvantages in certain contexts: Potentially large amounts of code, especially long-running routines, must be explicitly sprinkled with finalizer invocations. But such code must not be called in contexts in which finalizer invocations are unsafe.

Another potential use for an explicit call to run finalizers is to reclaim certain system resources which are particularly scarce. For example, it is common to force a garbage collection and invoke finalizers when a file open call would otherwise fail due to lack of file descriptors. However, this is complicated by several issues, which often seem to be overlooked:

1. Running one finalizer may cause other objects to be eligible for finalization. For example, a buffered file may need to be finalized (and flushed) before the underlying raw file can be finalized (and the descriptor reclaimed). In the case of Modula-3 style ordered finalization, the finalizers would implicitly be run in consecutive collection cycles. As we explain in appendix A, this behavior can, and often must be, emulated in Java.

2. The resource allocation call, *e.g.* the file open call, may be made from client code that holds locks. These locks may be needed by finalizers.

The first issue is easily addressed with careful interface design. Rather than attempting to run all finalizers with

```
System.gc();
System.runFinalization();
```

we should use

```
do {
  System.gc();
  System.runFinalization();
} while (<resource unavailable>
      && <System.runFinalization() did something>)
```

The second issue is more difficult to resolve.

Assume that `System.runFinalization()` is called from a thread holding lock L and no other locks. It is possible to, at least logically, run each finalizer in its own thread. This will cause exactly those finalizers that do not need L to run to completion. That's perhaps the best we can do, but it certainly leaves open the possibility that some blocked finalizers would have caused other, possibly more interesting, objects to be dropped.

A second possibility would be to insist that `runFinalization()` not be called from contexts that might hold locks shared by finalizers. That would require similar restrictions on file open or other resource allocation routines, which might call `runFinalization()` internally. This appears to us to be a viable alternative.

[19] See for example a typical complaint at **http://www.geocrawler.com/archives/3/196/1997/9/0/1089518/** or the more detailed discussion at **http://www.cs.arizona.edu/sumatra/hallofshame/monitor-finalizer.html** The latter overlooks the requirement on java.long.Object.finalize that "the thread that invokes `finalize` will not be holding any user-visible synchronization locks ...", and thus confuses an implementation bug for a specification bug. The GNU Java compiler corrected a similar implementation bug after we pointed out the problem. We believe this was a common mistake in this context, which has contributed to the bad reputation of finalizers.

[20] __del__ methods in Python

The current Java API specification[15] is unfortunately unclear on the intended usage model, or even whether it is acceptable to run finalizers in the thread invoking `runFinalization()`.

We conclude that it is probably useful to explicitly invoke finalizers to reclaim needed resources in this manner. We are aware of several Java implementations that implicitly try to to do so. But it is not clear to us that any of these systems do so completely safely, or through the right interfaces.

4. CONCLUSIONS

We have pointed out that C++ destructors and Java finalizers are completely different facilities. C++ destructors are used to provide guaranteed cleanup actions at well-defined program points, especially in the presence of exceptions. In our view[21] they are more closely related to Java *synchronized* and `try { ... } finally { ...}` blocks or, in the case of locks, *synchronized* blocks, than they are to Java `finalize` methods.

In contrast, finalization in languages like Java is necessary in order to manage resources other than garbage-collected memory based on garbage-collector-discovered reachability. Without this facility, it would often be necessary to basically redo the garbage collectors work in order to manage these resources. Although it has clearly introduced a significant amount of confusion, a properly designed facility can be used safely, and does not add significant complexity beyond that inherent in multi-threaded programming.

Although we believe that finalization is an essential facility in many large systems, we do not believe that it should be used frequently. Based on the limited statistics we have seen, one use per 10,000 lines or more of well-written code is probably typical. But eliminating use of finalization would touch nearly every module in such systems. Even the small amount of complexity inherent in finalization can normally be isolated to a few modules of a large system. This is again different from normal C++ destructor usage, which tends to be far more pervasive, while each individual use tends to be far less essential.

The issues for application programmers center on the fact that finalization effectively introduces an additional thread of control, and thus concurrency issues must be considered, even for otherwise single-threaded applications. Arguably no fundamentally new issues are introduced, but the importance of understanding concurrency issues is elevated. Synchronization is essential for finalizers.

Language implementors must also respect the fact that finalizers naturally constitute a separate asynchronous thread of control.[22] This applies to any finalizers on heap objects, whenever these become inaccessible at a point that is not completely apparent to the programmer. In particular, finalizers should never be run implicitly as part of a client thread which may be holding other locks. As was known to (some of) the Java community, this applies if the objects are managed by an automatic tracing garbage collector. But it also applies to semi-automatic (*e.g.* manually reference counted) memory management schemes, since in these systems the programmer has also chosen to abstract away the details about when objects become inaccessible. It is otherwise impractical to ensure that finalizers cannot be run inside a client holding locks.

More specifically, language designs supporting finalizers should ensure the following:

- In a multi-threaded environment, it must be guaranteed that finalizers will run in a thread in which no locks are held. Typically this means that either finalizers are run in their own thread(s), or that finalization-ready objects are enqueued and then run explicitly from programmer-initiated threads. Thus finalizers must be allowed to, and encouraged to, acquire locks.

- In purely single-threaded environments the programmer must be given explicit control over when to run finalizers. Typically this will be accomplished by explicitly enqueueing finalization-ready objects, as in [5].

- The language specification must provide the programmer with a way to ensure that objects remain reachable long enough to prevent premature finalization. The last run-time representation of a pointer to an object may disappear long before the last logical access to one of its fields. There seem to be clean ways to provide the necessary guarantees, but it does not appear to us that any language specification currently does so.

- Library calls to explicitly invoke the garbage collector and finalizers are apparently quite useful in managing relatively scarce non-memory resources. However primitives such as Java's *System.runFinalization* must be clearly specified with respect to synchronization behavior, and they must accommodate the fact that there may be dependencies among finalizers. Current language specifications generally appear to fail on both counts.

As far as finalization is concerned, there is no qualitative reason to prefer a reference-counting collector over a tracing collector. A reference counting collector may have a quantitative advantage in that it may defer their execution for a shorter period of time. But their execution must still be deferred, at least in some cases.

5. ACKNOWLEDGEMENTS

The observations about the necessity for synchronization in finalizers grew out of discussions, notably with Barry Hayes, many years ago. The problem, though not the solution, is partially outlined in [9].

Some of the conclusions here grew out of discussions on the gcj (GNU Java compiler) mailing list. Notably Andrew Haley[23] pointed out the danger in calling runFinalization from a thread holding locks, *e.g.* while trying to allocate a file descriptor. (Several bug reports on Sun's web site apparently made similar observations somewhat earlier, though not known to the author at the time.)

Guy Steele contributed to several discussions on finalizer ordering in Java, which are partially reflected here. The anonymous reviewers provided many useful suggestions.

[21] A similar point is made in *e.g.* [1]

[22] It must be asynchronous in the sense that the finalizer cannot always run at the specific point at which a thread discovers the object to be inaccessible. This does not by itself preclude cooperative multi-threading, with potentially deterministic execution.

[23] See **http://gcc.gnu.org/ml/java/2001-12/msg00390.html**.

6. REFERENCES

[1] J. J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[2] H.-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[3] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Software Practice and Experience*, 25(12):1315–1330, December 1995.

[4] G. Colvin, B. Dawes, P. Dimov, and D. Adler. Boost smart pointer library. http://www.boost.org/libs/smart_ptr/.

[5] R. K. Dybvig, C. Bruggeman, and D. Eby. Guardians in a generation-based garbage collector. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 207–216, June 1993.

[6] ECMA. *Standard ECMA-334: C# Language Specification*. ECMA, December 2001.

[7] J. R. Ellis and D. L. Detlefs. Safe, efficient garbage collection for C++. Technical Report CSL-93-4, Xerox Palo Alto Research Center, September 1993.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[9] B. Hayes. Finalization in the collector interface. In *International Workshop on Memory Management (IWMM 92, LNCS 637)*, pages 277–298, 1992.

[10] J. Horning, B. Kalsow, P. McJones, and G. Nelson. Some useful modula-3 interfaces. Technical Report 113, Digital Systems Research Center, December 1993.

[11] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.

[12] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[13] P. Rovner. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, July 1985.

[14] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[15] Sun Microsystems. Java 2 platform, standard edition, v 1.4.0 api specification. http://java.sun.com/j2se/1.4/docs/api/, 2002.

[16] G. van Rossum. Python reference manual. http://www.python.org/doc/current/ref/ref.html.

A. APPENDIX: FINALIZER ORDERING

Java finalizers and C# "destructors" differ from the Modula-3/Cedar approach in two important ways:

1. Objects may be finalized even if they are reachable from *other* finalization-enabled objects. Thus Java objects with nonempty *finalize* methods must explicitly take care to keep objects they need accessible through some other path.
2. In C#, a reachable object, *e.g.* an object direct referenced from a static member of a class may be finalized at program termination. Java had a facility to do the same, though that has since been deprecated.

Before we discuss these, we can observe that finalizers generally fall into two categories: Those that simply deallocate operating system resources, and those that touch other user data structures. A useful finalizer must generally do one of these: Only touching the object itself is useless, since it is no longer reachable, and thus can't be seen by the rest of the program.

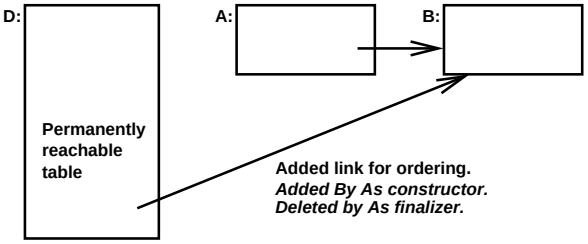

Figure 4: Enforcing finalizer dependence of A on B

The above changes don't affect the simple deallocation of system resources. For other kinds of finalizers, both changes add complications, in that a finalizer can no longer assume that the objects it needs have not yet been finalized.

We believe that change (2) above is a defect that has been remedied in Java with the deprecation of $System.runFinalizersOnExit$, and should be repaired in C#. For a well-designed operating system it is typically not a significant improvement for finalizers that only return resources to the operating system, *e.g.* by closing file descriptors. Operating systems generally reclaim those on process exit anyway to avoid resource leaks when programs crash. But in this environment we do not know how to write reliable finalizers that do much more than this: There can no longer be a guarantee that any other objects, even if they are clearly reachable through ordinary roots, have not been finalized before a given finalizer runs.[24]

Even if a finalizer simply wants to generate output to the standard error stream it is hard to see how it can guarantee that it will not have been previously closed. Since the existence of a finalizer for a particular class is generally viewed as a private implementation detail for that class, every finalizer would have to be prepared for every object it touches to have already been invalidated. It is hard to see how it would be practical to write finalizers under this kind of assumption. The situation is aggravated in that any errors along these lines are likely to result in intermittent symptoms.

We believe that change (2) was motivated by a desire to guarantee that finalizers run eventually to enable them to be used for tasks such as removing temporary files. We showed in our last example in section 3.2.4 that this can be done correctly without this change.

Change (1) is more interesting, especially since it has persisted in Java. It requires that if the finalizer for object A depends on another object B, which may also have a finalizer, object A will generally need to explicitly ensure that object B remains accessible, where in the Modula-3 case the collector implicitly handled the ordering.

This is possible by simply having A's constructor add B to a reachable data structure D, and then having A's finalizer remove the reference[25] as is shown in figure 4.

[24] As is is pointed out in the Java API documentation [15], the real reason for deprecating this call in Java was apparently only mildly related to this. If other threads are still running (as is likely in the presence of daemon threads), there is also no way to guarantee that finalizers won't be called on objects that are still being actively used.

[25] This relies on our earlier assumption about reachability.

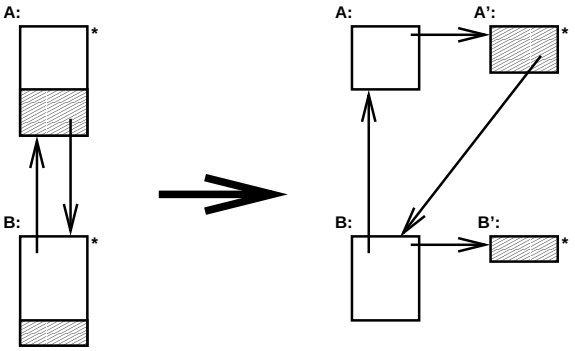

Figure 5: Elimination of finalization cycle

Change (1) does have the advantage that long lists or cycles of finalization-ready objects can automatically be finalized in a single cycle. But the same effect can usually be achieved with Modula-3 style finalizers by breaking objects into pieces, one of which contains only those fields accessed by the finalizer. Consider again the case in which A's finalizer needs access to B. Now assume also that B holds a reference to A, which is *not* needed by B's finalizer. This can be transformed as in figure 5.[26] Here fields needed by finalizers have been indicated by shading, and "*" indicates that an object is finalization-enabled.[27] In the transformed version, A' is not reachable from a finalization-enabled object, and can thus be finalized first. On the other hand, B' is reachable from A' and must therefore wait for B''s finalization.

In general, for Modula-3 style finalization, the programmer should keep finalization-enabled objects small, and avoid references in finalization-enabled objects that are not followed by finalizers. If this rule is followed, we believe that neither cycles nor long chains of objects are a serious issue. (Any remaining cycles between finalization-ready objects are easily detectable by the runtime.)

B. APPENDIX: SYNCHRONOUS FINALIZATION FAILS

We illustrate how synchronous finalization, such as one might obtain with a C++ reference counting collector that naively uses C++ destruction to implement finalization, can unexpectedly deadlock. A "runnable" version of this example in C++, using Boost `shared_ptr` as the "garbage collector" and Boost synchronization can be found at **http://www.hpl.hp.com/personal/Hans_Boehm/popl03/c++example**.

Consider adding the following to the "external object data" example from section 3.2.3. The call $c.update(x)$ updates the information stored in the *impls* array for c, based on some information associated with x. If c's representation is not shared, the update is performed in place. Otherwise, it is first copied:

```
public synchronized void update(C other) {
  synchronized(impls) {
    int count = impl_use_count[my_index];
    T new_val =
        combine(impls[my_index].data,
                X.messy_fn(other));
    if (count > 1) {
      // Clone my C_impl.
        int new_index = first_available();
        impl_use_count[new_index] = 1;
        impls[new_index] = new C_impl(new_val);
        impl_use_count[my_index] = count - 1;
        my_index = new_index;
    } else {
      impls[my_index].data = new_val;
    }
  }
}
```

Assume that

1. $X.messy\_fn$ computes some property of its class C argument, which is expensive to compute. This property never changes for a given C instance.
2. X was written by someone else whom we haven't heard from in five years.
3. Unknown to us, X maintains a small cache of C values it was recently passed, together with the corresponding return values. When the cache becomes too large, an old value may be dropped by being overwritten.

Recall that so far we assume that locks cannot be reacquired by a thread. Now consider the scenario in which

1. We call *update*.
2. It acquires the lock, reads a *count* of 2, and then calls $X.messy\_fn$.
3. $X.messy\_fn$'s cache does not yet contain *other*. It thus computes the result and replaces a previous cache entry c.
4. c happened to share a C_impl with the entry being updated.
5. c becomes eligible for finalization.

With a simple reference counting garbage collector (*e.g* implemented like Boost `shared_ptr` [4]) the Scenario continues:

1. Finalizers are invoked immediately when a zero reference count is detected, from the thread that caused the reference count to be decremented.
2. c's finalizer will be invoked from the thread that called *update*.
3. This thread still holds the lock for class *impl*. Deadlock!

The current Java Language Specification[8] does not guarantee this to be correct. Since the reference from D to B is never actually followed from a non-finalizer thread, there is no guarantee that the collector must consider it when determining reachability.

[26] As in the previous construction, we are assuming that object reachability is defined so that A' is not finalized while A is accessible.

[27] We have also assumed that, as in Modula-3, a finalization-enabled object may be finalized even if it points to itself. This is not the default for our collector, though perhaps it should be. Otherwise, and in some more complex cases, it might be necessary to expose B' and to make A' reference B' directly to make it apparent to the GC that there is no real reference cycle.

If instead we assume Java lock semantics, we may end up with data structure corruption instead of deadlock. Instead of the above, the scenario continues:

1. *c*'s finalizer will be invoked from the thread that called *update*.
2. The finalizer reacquires the lock, ignoring the fact that we were already in the middle of updating the underlying data structure.
3. The finalizer decrements the count (but not *update*'s local copy).
4. *Update* installs a count of 1 into the old entry, when it should have been zero.

Clearly, with full insight into the entire system, we could easily program around this problem (though it may be harder for more realistic examples). But the code, as written, *should run correctly*. The programmer correctly used locks to ensure mutual exclusion, and correctly counted on the garbage collector to take care of memory deallocation and finalization behind the scenes. In order to fix the problem, we need to understand that the collector will potentially invoke a finalizer for a C object as part of an invocation of $messy\_fn$. But the whole purpose of the collector was to make it unnecessary to reason about deallocation timing in this way.

The only way to preserve the programmer's abstraction is to decouple the finalizer invocation from the thread that happened to drop the last object reference, i.e. to run the finalizer asynchronously, effectively in its own thread. Thus even if we use a simple reference counting collector, which detects inaccessibility immediately, we should still enqueue the finalization call and carry it out later in a different thread.