Historical notes on module unloading

in the Oberon operating system

Andreas Pirklbauer

1.6.2019

Purpose

This note describes how *module unloading* is handled in various implementations of the Oberon operating system[1].

Module unloading in the Oberon operating system

In the Oberon system, there exist three possible types of references to a loaded module M[2]:

1. *Client references* exist when other loaded modules *import* module M.

2. *Type references* exist when type tags (addresses of type descriptors) in *dynamic* objects reachable by other loaded modules refer to descriptors of types *declared* in module M.

3. *Procedure variable references* exist when procedure variables in *static* or *dynamic* objects reachable by other loaded modules refer to procedures *declared* in module M.

Interpretations of the *semantics* of module unloading can be broadly grouped into two main categories: (a) schemes that explicitly allow invalidating future references, and (b) schemes where all past and future references must remain unaffected at all times.

a. Schemes that explicitly allow invalidating future references

Schemes that explicitly allow invalidating future references typically only check *client* references prior to module unloading. Consequently, unloading a module from memory *may*, and in general *will*, lead to "dangling" *type* and *procedure variable* references pointing to module data that is no longer valid[3].

---

[1] *http://www.projectoberon.com*

[2] *An Oberon module can be viewed as a container of types, variables and procedures. Types can be declared globally (in which case they can be exported and referenced by name in client modules) or local to a procedure (in which case they cannot be exported). Variables can be declared as global variables (allocated in the module area when a module is loaded) or as local variables (allocated on the stack when a procedure is called). Anonymous variables with no explicit name declared in the program can be dynamically allocated in the heap via the predefined procedure NEW. Procedures can be declared as global or local procedures, and can be assigned to procedure variables. Thus, in general there can be type, variable, procedure and procedure variable references from static or dynamic objects of other modules to static or dynamic objects of the modules to be unloaded. However, only dynamic type references and static and dynamic procedure variable references need to be checked during module unloading for the following reasons: First, static type, variable or procedure references from other modules can only refer by name to types, variables or procedures declared in the modules to be unloaded. Such references are already handled via their import/export relationship during module unloading (if clients exist, a module or module group is never unloaded) and therefore don't need to be checked separately. Second, dynamic pointer references from global or dynamic pointer variables of other modules to dynamic objects reachable by the modules to be unloaded should not be checked, as they should not prevent module unloading. In the Oberon system, such references will be handled by the garbage collector during a future garbage collection cycle, i.e. heap records reachable by the just unloaded modules and other still loaded modules will not be collected, whereas heap records that were reachable only by the unloaded modules will be collected – as they should. Thus, the handling of pointer references is delegated to the garbage collector. Finally, pointer variable references to statically declared objects are only possible by resorting to low-level facilities and should be avoided – and, in fact, be disallowed (pointers should point exclusively to anonymous variables allocated when needed during program execution).*

[3] *If the programming language Oberon-2 is used, there can also be references to method tables (which however are typically allocated within type descriptors).*

An important case is where a structure rooted in a variable of base type T declared in a base module M contains elements of an extension T' defined in a client module M'. Such elements typically contain both *type* references (type tags) and *procedure variable* references (handler procedures) referring to module M'. This is common in the Oberon viewer system, for example, where M is module *Viewers*. If the client module M' is unloaded and the module block previously occupied by it is overwritten by another module loaded later, any still existing references to M' become *invalid* at that moment. Global procedure variables declared in other loaded modules may also refer to procedures declared in module M', although this case is less common (global procedure variables tend to be used mainly for procedures declared in the same module).

A wide variety of approaches have been employed in various implementations of the Oberon system to cope with the introduced dangling *type* or *procedure variable* references:

1. On systems that use a *memory management unit* to perform virtual memory management, such as Ceres-1 or Ceres-2, one approach is to *unmap* the module space of an unloaded module from virtual memory, thereby *invalidating* future references to it. When this is done, any still existing *type* or a *procedure variable* references point to a now *unallocated* page, and consequently any attempt to access such a page later will result in a *trap*.

   We consider this an unfortunate proposal for several reasons. First, users generally have no way of knowing *whether* it is in fact safe to unload a module, yet they are allowed to do so. Second, after having unloaded it, they still don't know whether references from other loaded modules still exist – until a *trap* occurs. But then it is usually too late. While the trap itself will actually *prevent* a system crash as intended[4], the user may *still* need to restart the system in order to recover an environment without any "frozen" parts, for example displayed viewers that may have been opened by the unloaded module. Note also that this solution requires special hardware support, which may not be available on all systems.

2. On systems that do *not* use virtual memory, such as Ceres-3 or FPGA Oberon on RISC, the easiest way to cope with dangling references is to simply *ignore* them, i.e. to *always* release the memory associated with a module without any further precautions (unless clients exist). While an attempt to access an unloaded module goes undetected *initially*, the system is still left in an *unsafe* state. It will become *unstable* the very moment another module loaded later *overwrites* the previously released module block *and* other (loaded) modules still refer to its *type descriptors* or *procedures*. This is of course undesirable.

3. Another approach, which however can be used only for *procedure variable* references, is to identify *all* such references and make them refer to a *dummy* procedure, thereby preventing a run-time error when such "fixed up" procedures are called later. Of course, this solution requires the system to *know* the locations of all procedure variables in all static and dynamic objects at run time[5]. It was used in an earlier version of Experimental Oberon, but was later discarded, mainly because the resulting effect on the *overall* behavior of the system is essentially impossible to predict (or even detect) by the user. The fact that *some* procedure variables *somewhere* in the system no longer refer to *real* but to *dummy* procedures typically becomes "visible" only through the *absence* of some action – such as mouse tracking if the unloaded module contained a viewer handler, for instance.

---

[4] For example, if the unloaded module implements a subframe type, a trap is generated if the enclosing menu viewer attempts to send a "close" message to the subframe by calling its handler.
[5] This can be achieved by including the locations of procedure variables in the module block as follows: the offsets of global procedure variables in the module's data section are stored in a separate array in the module's meta data section, while the offsets of procedure variables in dynamically allocated records are added to the type descriptor associated with each such record.

4. On systems that use *indirection* for procedure calls via a so-called "link table", the same effect can be achieved by setting the *link table entries* for all referenced *procedures* of the module to be unloaded to *dummy* entries, rather than locating and modifying each individual procedure *call* that may exist anywhere in the system.

   Note that using a link table to implement indirection for procedure calls is only viable on systems that provide *efficient* hardware support for it. On such systems, an "address" of a procedure is not a real memory address, but an *index* to this translation table – which the caller consults for every procedure *call*, in order to obtain the actual memory location of the called procedure. Indirection for procedure calls via a link table was used in some of the earlier versions of the Oberon system on Ceres computers, which were based on the (now defunct) NS32000 processor. This processor featured a *call external procedure* instruction (CXP k, where k is the index of the link table entry of the called procedure), which sped up the process of calling external procedures significantly[6]. Later versions of the NS processor, however, internally re-implemented the *same* instruction using microcode, which negatively impacted its performance[7]. For this and a variety of other reasons, the *CXP k* instruction – and with it the *link table* – were no longer used in later versions of the Oberon system.

5. Finally, we note in passing that for *type* references, it is possible to determine at *compile* time, whether a module *may* lead to references from other modules at *run* time. The criteria is the following: if a module M' does *not* declare record types which are extensions T' of an imported type T, then records declared in M' *cannot* be inserted in a data structure rooted in a variable *v* of the imported type T – precisely *because* they are not extensions of T (in the Oberon programming language, an assignment *p := p'* is allowed only if the type of *p'* is the same as that of *p* or an extension of it).

   One *could* therefore introduce a rule that a module M can be safely dispensed *only* if it does *not* declare record types, which are extensions T' of an imported type T. The flip side of such a rule, however, is that modules that actually *do* declare such types can *never* be unloaded (unless, of course, other ways to safely unload such modules are implemented).

Even though most of these approaches have actually been realized in various implementations of the Oberon system, we consider none of them truly satisfactory. In our view, these schemes appear to only tinker with the symptoms of a problem that would not exist, if only one adopted the rule to *disallow* the removal (from memory) of still referenced module data.

The main issue appears to be that the moment one *allows* modules to release their associated memory even when references to them still exist from other loaded modules, the resulting *dangling* references must be dealt with *somehow* in order to prevent an almost certain system *crash*. However, *fixing up* or *invalidating* references will *always* remove essential information from the system. As a result, the run-time behavior of the modified system becomes *essentially unpredictable*, as other loaded modules may *critically* depend on the removed functionality. For example, unloading a module that contains an installed handler procedure of a *contents frame* may render it impossible to *close* the enclosing *menu viewer* that contains it, thereby leading to a system with "frozen" parts. A similar problem may occur with references to *type descriptors*, if they are not persisted in memory *after* unloading their associated modules.

---

[6] The use of the link table also increased code density considerably (as only 8 bits for the index instead of 32 bits for the full address were needed to address a procedure in every procedure call). In addition, the link table used by the CXP instruction allowed for an expedient linking process at load time (as there are far fewer conversions to be performed by the module loader – one for every referenced procedure instead of one for every procedure call) and also eliminated the need for a fixup list (list of the locations of all external procedure references to be fixed up by the module loader) in the object file. A disadvantage is, of course, the need for a (short) link table.

[7] The internal re-implementation of the CXP instruction using microcode in later versions of the NS processor followed the general industry trend of implementing only frequent, simple instructions directly with hardware, while interpreting more complex instructions using internal microcode. In general, with the advent of highly regular reduced instruction set computers (RISCs) in the mid 1980s and early 1990s, the trend towards offering microprocessors providing a smaller set of simple instructions, most of them executing in a single clock cycle, combined with fairly large banks of (fast) registers, continued (and does so to this date).

b. Schemes where all past and future references must remain unaffected at all times

The second possible interpretation of *unloading* a module consists of schemes where all past and future references *must* remain unaffected at all times. In such schemes, module unloading can be viewed as an implicit mandate to preserve "critical module data", as long as references to the unloaded module still exist.

1. One could of course simply exit the *unload* command with an error message, whenever such references are detected. The user, however, may then be "stuck" with modules that he can *never* unload because they are referenced by modules over which he has no explicit control.

2. But the mandate could also be fulfilled by persisting any still referenced module data to a "safe" location before unloading the associated module.

   For *type descriptors* referenced by *type tags* in dynamic records an easy solution exists: allocate them *outside* the module blocks in order to persist them beyond the lifetime of their associated module. One possibility is to allocate them in the *heap* at module load time[8]. This has been implemented in Ceres-Oberon, for instance. Note that this approach eliminates dangling *type* references altogether and therefore also the *need* to check for them at run time – as type descriptors are now unaffected by module unloading.

   For *procedures* referenced by *procedure variables* in either static or dynamic objects no such simple solution exists. If one doesn't want to invalidate or fix up procedure variable references (as outlined in the previous section), the only way to "persist" procedures is to persist the *entire* module (recall that procedures may also *access* global module data or *call* other procedures declared in the same module).

   We conclude that if one wants to address both type *and* procedure variable references, one *cannot* unload the module block from memory, as long as references to it still exist[9].

3. A straightforward way to automatically persist both type descriptors *and* procedures consists of *never* releasing the module block of a module once it has been loaded. Instead, when the user requests the unloading of a module, it is only removed from the *list* of loaded modules. This amounts to *renaming* the module, with the implication that a newer version of the same module (with the same name) can be reloaded again[10]. This approach has been chosen in MacOberon[11], for instance. Since the associated memory of a module is never released, the issue of dangling type or procedure variable references is avoided altogether, as they simply cannot exist. However, it can also lead to higher-than-necessary memory usage if a module is repeatedly loaded and unloaded (typical on *development* systems). Nevertheless, such an approach may be seen as adequate on *production* systems where module unloading is rare.

4. A *refinement* of the approach outlined above consists of *initially* removing a module from the list of loaded modules (as in MacOberon), but *in addition* releasing its associated memory *as soon as* there are no more type or procedure variable references to it. If this is done in an

[8] *Note that one cannot simply move type descriptors around in memory, as their addresses are (typically) used to implement type tests and type guards. By allocating them in the heap at module load time, one avoids the need to move them to a different location when a module is unloaded.*
[9] *Of course, "mixed" variants are also possible. For example, one could allocate type descriptors in the heap at module load time (as in Ceres-Oberon), and either fix up all procedure variable references or prevent the release of a module block if such references still exist; however, most modules referenced by type tags are also referenced by procedure variables – this is in fact the typical case for dynamic records containing installed handler procedures. Thus, it seems more natural to employ the same approach for both type and procedure variable references.*
[10] *In a specific implementation, one might choose to make the module completely anonymous or modify the module name such that one can no longer import it (e.g., by inserting an asterisk).*
[11] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*

automatic fashion (for example as part of a background process), module data is truly “kept in memory for exactly as long as necessary and removed from it as soon as possible”.

This is the approach chosen in Experimental Oberon. A variation of it was used in some versions of SparcOberon[12].

In sum, schemes were past and future references remain *unaffected* at all times avoid many of the complications that are inherent in schemes that explicitly *allow* invalidating references. A (small) price to pay is to keep modules loaded in memory, as long as references to them exist.

However, on *production* systems, there is typically no need to keep multiple copies of the same module in memory, while on *development* systems it is totally acceptable.

Finally, we note that on modern computers the amount of available memory, and therefore also the amount of *dynamic* data that may be allocated by modules, typically far exceeds the size of the module blocks holding the program code and global variables. Hence, not releasing module blocks immediately after a module *unload* operation typically has a rather negligible impact on overall memory usage.

* * *

[12] *http://e-collection.library.ethz.ch/eserv/eth:7103/eth-7103-01.pdf (SPARC-Oberon User's Guide and Implementation, 1990/1991)*