I've attached two solutions that provide details upon garbage collection and setting VSE memory sizes using the VirtualMachineConfiguration class.

The "Out of memory - unable to continue" message appears when VSE attempts to allocate memory but did not find enough available to do so. The error is usually fatal and does not provide useful information in the error.log file (contrary to what I had hoped).

There are some steps you can take to ensure you are not wasting the memory you allocate. These steps may not apply if you anticipate you are using most of the memory to allocate a large dataset:

1. Remove references to unreclaimed objects that are not garbage collected and should be.
These references occur when any object references or is referenced by a global, pool
dictionary, class variable, or similar global.

2. Attempt to reduce the number of windows or other Windows widgets referenced
all at once in the image. (Not likely a cause of concern since you mentioned your interface was
simple).

3. If you created a class that uses Windows system resources be sure to

implement a #release method to free those resources when the object is garbage collected. This #release method should invoke 'super release' to use the release methods of the superclasses.

4. If your image acts as a server with some high priority processes attempt to yield from these processes occasionally to allow the image time to perform its garbage collection.

Memory tuning.

Read solution 2444 carefully and also consider the following:

All memory space size parameters combined of VirtualMachineConfiguration must in total be less than 2GB. I've listed the default sizes for VirtualMachineConfiguration below. These defaults are listed in the VirtualMachineConfiguration class description in the Encyclopedia of Classes for Win32

arenaBytes 100MB
codeSpaceCommittedBytes 256kB
codeSpaceMaxBytes 1MB
flipSpaceBytes 256kB (must be same as newSpaceBytes)
newSpaceBytes 256kB

nominalFlipSpaceBytes 4096B
oldSpaceBytes 72MB
paddingSizeBytes 32768B

Engineering comments that they don't see the advantage of allocating as much as 2GB of
virtual memory since the performance is bound to be terrible once you run out of RAM
because of the high cost of paging.

If you exceed the 2GB allocation or choose inappropriate memory sizes you may see more unusual errors/exceptions in your application. For example, one customer would obtain "unable to allocate virtual address space" errors. He choose

```
maximum old space == 1072 MB.
maximum arena space == 1100 MB.
```

which exceeded the 2GB allocation.

Presuming that we need to impose a 2GB Windows virtual memory limit you need to adjust two parameters of VirtualMachineConfiguration correctly--arenaBytes and oldSpaceBytes. By default, arenaBytes is 100Mb and oldSpaceBytes is 72Mb. These values are interrelated and a rule of thumb would be to change them in the same proportion. I believe this means
that you should choose an old space value in bytes about 72% of the arena bytes.
There is some lattitude in this selection.

The following example shows how one might limit the total memory used by the VM
when it starts. Similar coding should be evaluated to increase the total memory:
--
Here are some details on using the VirtualMachineConfiguration class to limit the amount of memory allocated by the VM. You need to adjust two parameters of VirtualMachineConfiguration to impose this limit correctly--arenaBytes and oldSpaceBytes. By default, arenaBytes is 100Mb and oldSpaceBytes is 72Mb. These values are interrelated and should be changed in the same proportion. Here is a code sample that changes the limit of arenaBytes from 100Mb to 60Mb and oldSpaceBytes to 25Mb less than the 60Mb limit.

```
| config bytes |
  config := VirtualMachineConfiguration fromImage. "Read from the
image"
  bytes := 60000000.
  config arenaBytes: bytes.
  config oldSpaceBytes: ( bytes - 25000000 ).
```

```
    config toImage. "Write to the image"
--
```

Solution 2481

Garbage Collection in Smalltalk/V
by David Ornstein

Overview

Smalltalk frees developers from thinking about memory management. This is not trivial. In applications that must be explicitly manage their own memory, up to a quarter of the code may relate to memory management. That's code that someone has to write. It's code that will have bugs and must be tested. Simply stated, the less code you can have, the smoother your project will go.

Especially when compared with many other languages, Smalltalk's automatic memory management is a great benefit. You can put more of your attention where it belongs: in the behavior of the application you are building. It would be nice if this meant that you never have to worry about memory management. It doesn't. It does mean though, that you have to worry about it much less often.

Smalltalk's object allocation and deallocation model is very simple. Imagine a chunk of memory that holds all objects. To create a new object, Smalltalk searches for empty space in this chunk. If the chunk does not have enough empty space to hold the new object, the magic begins. A part of Smalltalk called the garbage collector looks through all the objects in the chunk and declares that all objects that are not referenced from any other object are garbage and may be discarded. All the objects that survive this project are compressed into one end of the chunk, and the empty space accumulates at the other end (see figure 1 - Garbage Collection). Now the original request to allocate an object continues where it left off, but this time succeeds.

Figure 1 - Garbage Collection

The garbage collection process is the only thing that causes an object to be deallocated. It's quite a leap of faith that a simple rule ("unrefferenced objects are garbage") will work, but it does. It lets you build your applications blissfully unaware of memory management issues -- most of the time. As I hinted above, and discuss in more detail below, you still have to be involved occasionally.

This article discusses how the garbage collection process works, and when, why and how to monitor and change it's behavior. While this article focuses on Smalltalk/V Win32 version 2.0, almost all of it applies to the other 2.0 releases. At the end of an article, it says a bit about the upcoming Visual Smalltalk release (a.k.a. 3.0).

Some parts of this article describe logically what's going on, but don't explain the real implementation. The goal is to help you understand garbage collection and it's impact on you as an application author and Smalltalk programmer, not as a Smalltalk system implementor.

Spaces: The Final Frontier?

If you look closely at the objects that a Smalltalk application creates, you'll notice that most objects live for a very short time, a very long time or forever. Objects such as temporary variables live for a short time, perhaps a minute or less. Most objects live a moderately long to long lifetime (minutes to hours). Some objects, for example those in DLLs1, live forever (the garbage collector never collects them). This division suggests a way to organize the garbage collection architecture.

To take advantage of these different object lifetimes, Smalltalk divides it's object space into multiple physical spaces. The first of these spaces is called new space, and it holds all of the objects that live for a very short time. The second space is called old space, and holds most of the rest of the objects. Finally, DLL space contains all the objects loaded from object libraries (DLLs). The system collects garbage in each space independently. The big saving comes because the system can collect garbage in new space without spending time to look through all the objects in other spaces.

What does garbage collection look like with these object spaces?

When the system creates a new object (for example when you evaluate Ordered Collection new), it creates the object in new space. (If the object is too large to fit in the new space, the system creates the object in old space). If there is enough contiguous free memory at the top end of new space, the system uses a slice of that memory. If there isn't enough of free memory, the system garbage collects new space using an operation called flip. All referenced objects in new space are copied to the bottom end of new space, overwriting all the unrefferenced objects and making space for new object allocation. Flips happen every few seconds and they are so fast (<100ms) that you never notice them.

The garbage collector tracks the age of all objects in new space. A new object's age is zero. Each flip increases the objects age by one. When an object survives a certain number of flips, it isn't new anymore and the garbage collector moves it to old space. This process is called tenuring and happens during the flip. This is the bridge between new space and old space (see Figure 2 - Object Spaces).

Figure 2 -- Object Spaces

When the system needs to put an object into old space, it looks for free memory at the top end of old space. If it finds it, it uses it. If not, it can satisfy the request in one of two ways: making old space bigger or by garbage collecting old space.

To cooperate with other running applications, Smalltalk limits how big old space may become before it invokes the garbage collector. If old space is smaller than it's threshold, Smalltalk asks the operating system for more memory and adds that memory to the end of old space.

If the old space reaches the size threshold, the system garbage collects old space. This operation is called a compact. A compact is the only garbage collection operation that might last long enough for the user to notice2 . Again the system overwrites unreferenced objects to the bottom end of old space. It then releases most of the freed memory back to the operating system, keeping a handful of pages around to satisfy future allocation requests.

Objects loaded into the image from an object library (DLL) are loaded into their own space. The system doesn't allocate objects in DLL space, it just binds DLLs into DLL space. This means that the garbage collector never really collects garbage in DLL space.

Peeking Behind the Curtain

Sometimes the best way to understand a subject as complicated as this is to see it in action. This section shows you how you can monitor what the garbage collector is doing.

The first thing to know is that you can cause a compact to occur anytime, without creating any objects, by evaluating 'Smalltalk unusedMemory'. This message provides information about how much memory is free. Under an operating system that supports virtual memory (as both Windows and OS/2 do), the answer to #unusedMemory is not usually very interesting. What is interesting is that giving an even remotely correct answer requires that the system perform a compact. You use this technique with the others below to see the result of a compact.

The ProcessScheduler class provides some hooks so that you can monitor the garbage collector. There is one instance of this class in the system and it is stored in the global called Processor. ProcessScheduler has a number of instance variables that hold information about garbage collection (see Table 1--ProcessScheduler status variables). Processor updates these values when you send it the message #status.

Variable	Description
bytes collected	The number of bytes of garbage that were collected during the last compact.
bytes flipped	The number of bytes that survived the last flip.
bytes tenured	The number of bytes that were moved from new space to old space during the last flip
new space pages	The number of pages3 of memory allocated to new space
old space pages	The number of pages of memory allocated to old spaces.

Table 1- Process Scheduler status Variables

When a compact happens, the system sends the message #compact to Processor. You can monitor compacts by adding the following code to the end of the ProcessScheduler>>compact method:

```
self status.
Transcript nextPutAll: 'compact',
    bytesCollected printString,
Transcript nextPutAll: 'old space now',
    oldSpacePages printString,
    ' pages - next compact at ',
    oldSpaceThreshold printString,
    ' pages)';
    cr.
```

Similarly, when a flip happens, the system sends the message #flip to the Processor. You can monitor flips by adding the following code at the end of the Process>>flip method:

```
self status.
Transcript nextPutAll: 'Flip',
    bytesFlipped printString,
```

```
        ' bytes flipped and ',
        bytesTenured printString,
        ' bytes tenured ',
Transcript nextPutAll:  ' (about',
        ((newSpacePages * 4096) - bytesFlipped)
        printString, ' bytes discarded) ' ;
        cr.
```

Once you have made these changes, try something simple like bringing up a class hierarchy browser and watch the Transcript. You should see a number of flips happen. Evaluate 'Smalltalk unusedMemory' to see the impact of a compact. Evaluate the following code:

```
Smalltalk unusedMemory.
Temp  : = String new: 500000
Smalltalk unusedMemory.
Temp  : = nil.
Smalltalk unusedMemory
```

You should notice that old space grows dramatically after allocating the half - megabyte string. After releasing the only reference to the string (by putting a nil in Temp), the final compact operation frees the memory that the string occupied and old space drops to its original size.

The Root of your Troubles

The garbage collector uses a simple rule: unreferenced objects are garbage. Because of this rule you need to watch out for unintentional object references. Because an object may be the root of a tree of objects, an unintentional reference to one object can prevent the system from discarding thousands of objects.

To analyze unintentional references you follow the chain of references backwards from the suspect object until you understand why it hasn't been discarded. The chain of references will eventually lead back to one of the following places:

* Global Variables - this is pretty obvious!
* Class variables - remember that objects stored in class variables stay around until you change the class variable
* Dependents (the class variable in class object) - this is very common. If you forget to release a dependency, the object which has the dependent will stay (erroneously) listed in the dependents
* Notifier - this is the sole instance of NotificationManager and holds references to all open windows. Under some circumstances, references can erroneously remain from this object to invalid (closed) windows.
* Class instance variables - for those systems that support them.
* Workspace variables - for those systems that support them.

You use the message Object>>allReferences to follow the chain of references. This message answers an Array of all objects that reference to the receiver. You successively use #allReferences to find your way back to the start of the chain. If you use inspectors to follow the chain, the inspectors will have references to the objects on the chain. You should ignore these extraneous references.

If you suspect that a class has instances when it shouldn't, you can send #allInstances to the class to get an array of all instances of the class. If you find objects in the array, you can use allReferences to find out why they aren't garbage.

Taking Control

A running application can modify the behavior of the garbage collector. Tuning a garbage collector, however, is a black art: you're more likely to harm a systems performance than improve it. Treat the information in this section like a sharp knife: it's important to have, but if you're not careful, you'll hurt yourself.

You can adjust two aspects of the garbage collector: the old space threshold and the tenuring policy (see Table two - ProcessScheduler configurable variables).

Variable	Description
old space Threshold	The number of pages that old space may grow to before a compact operation is forced
tenure policy	The number of flips that an object must survive before it is moved (i.e. tenured) to old space.

Table 2 - ProcessScheduler configurable variables

To adjust a parameter, assign the desired value to the variable and then send #configure to the ProcessScheduler

The main purpose for adjusting the old space threshold is to defer an upcoming compact operation. For example, a compact would be intrusive if it happened while a window was opening. To avoid this you adjust the old space threshold up by a small safety margin, ensuring that old space can grow by a margin before a compact happens. Below is a method that will make this adjustment. You can invoke it when you want to avoid an upcoming compact:

```
ProcessScheduler>>avoidCompact
    | margin |
    self status.
    margin : = 10.
     (oldSpaceThreshold - old SpacePages) <
        margin ifTrue: [
        oldSpaceThreshold : =
              oldSpacePages + margin.
        self configure
        ]
```

In this example, the margin is 10 pages of memory.

The usual sign that you should change your tenuring policy is that many objects are surviving just long enough to make it into old space and then becoming garbage. To avoid cluttering up old space (the whole point to having a new space in the first place!), you can increase the tenuring policy.

Adjusting the tenuring policy is a complex undertaking. If you are going to change it, do so only in those parts of your application where you can carefully study the impact of the change. Don't assume that because it improves the performance of one part of the application that it will have the same effect elsewhere.

Visual Smalltalk

While the new Visual Smalltalk products (version 3.0) introduce some changes in garbage collection, most of this article applies equally to the new version. Here's a brief look at some of the changes:

* Configuring initial sizes in the image file. The initial sizes and thresholds for the object spaces can be specified in the image file as well as changed in the running image.
* Finalization. In some applications, it is useful to be able to be notified when an object is being discarded. Visual Smalltalk supports this capability with a feature called finalization.
* Reduce tenuring. A new tuning parameter has been added to the garbage collector that, for some applications can reduce compacts by tenuring fewer objects.
* ProcessScheduler accessing methods. Accessing the information in class ProcessScheduler is easier because Visual Smalltalk provides accessing methods on the class.
* ProcessScheduler events. The ProcessScheduler class now raises events for compact and flip operations, allowing external monitoring without having to change the code in the ProcessScheduler itself.

Solution 2444

Chronicles of the Virtual Machine
Memory Usage on Visual Smalltalk: Problems, Concerns and Possible Solutions.
By Roger Thayer

Frequently users of Visual Smalltalk want to know about memory usage, and what they can do to improve the performance and/or stability of their application or develepoment environment. This requires an understanding of how memory is used in Visual Smalltalk.

There are three kinds of memory spaces in Visual Smalltalk virtual machine (VM). The simplest is library space, where objects in SLLs are loaded. These objects never move. If an SLL is unbound, the memory for the library will be decommitted.

Any new objects that are created by Smalltalk that survive long enough are in oldSpace. When you
save your image, most of the v.exe file is a representation of old space. OldSpace is only as big as
it needs to be, and its size is adjusted on every compact. Compacts are needed to do some virtual machine operations, like #saveImage and #allInstances, but otherwise it is controlled by Smalltalk. Objects are "tenured" into oldSpace from newSpace.

NewSpace is where objects are created. The size of newSpace is setable, but it is constant once the image is started. NewSpace has a twin, called flipSpace, which is using during a flip (a garbage collection of newSpace) to copy all live objects to. FlipSpace is only as big as the amount of space needed to hold all the live objects in

newSpace, and its size is adjusted after every flip. Any object that survives two flips is eligible to be tenured to oldSpace. Whether it is actually tenured is based on the nominalFlipSize, which effectively controls how much space can be held in newSpace before anything can be tenured. A nominalFlipSize of zero means every object that survives two flips will be tenured. The higher the nominalFlipSize, the more objects will stay in newSpace and not be tenured.

If users want to get an idea of what is happening in their system, they can look at some of the instance variables of the Processor, which is a global instance of a ProcessScheduler. Every garbage collect, either a flip or compact, causes the ProcessScheduler method #rebalance to be executed. In this method, you can look at the varibales..
1) bytesFlipped: The number of bytes that survived the flip and were not tenured.
2) bytesTenured: The number of bytes that were moved from newSpace to oldSpace.
3) oldSpacePages: When muliplied by 4096, this is number of bytes oldSpace takes up.
These values can either be written to the screen or written to a log file.

Now lets consider possible problems a user might have.
1) We seem to be spending too much time swapping (in version 3.0 or higher).
It may be that they are using so much memory that the OS is forced to swap, but there are things to check. The swapping may be caused by memory compacts, and maybe compacts are being done frequently.
To find out, a bell can added to Process class>>gcInterrupt, which happens after every compact.
If compacts are being done often, the user should monitor the growth of oldSpace by checking the values of oldSpacePages, as described above. The user may notice that many objects are being tenured, but the flipper rarely, if ever, flips more than 100K, for example. In this case, he could use the VirtualMachineConfiguration class to set the nominalFlipSize to be 100K. This should greatly reduce the amount of tenuring being done, and therefore, the compacts. If it doesn't, that implies that a large amount of objects are living a long time, possibly forever. At this point, the user needs to make sure he doesn't have global or class variables hanging onto objects he no longer needs. If the data suggests that the nominalFlipSize needs to be very large, then newspace can be enlarged from its 256K default by using the VirtualMachineConfiguration class. Another route for developers of large applications is to bind and unbind portions of their applicaion. This is much more work, since they have to separate their app into logical components, but the benefits can be great. In fact, it may be the only solution for some.

2) We sometimes crash when low on memory.
Despite the fact that Smalltalk has preallocated large portions of memory, it is not guaranteed that when
it wants to actually commit and use that memory, the memory will be available. To help safeguard this situation, Smalltalk keeps a padding of memory that it uses a buffer so that it hopefully won't run out. The bigger this buffer, the safer Smalltalk is, but Smalltalk's footprint increases by that amount. For that reason, the default buffer is pretty small, 32K. If low memory situations are a problem for the user, this

number should be much bigger, determined by experimentation, since it varies from app to app. A reasonable first try would be 256K. The size can be changed by using the VirtualMachineConfiguration class (in version 3.0 or higher).

3) Why does Smalltalk run out of memory when virtual memory should be infinite?
First of all, we have noticed that when RAM is tight, the current implementations of virtual memory are somewhat unstable. However, if you run out of memory and you have plenty of RAM and hard disk space, it could be because our default old space limit was hit. However, this is very large, 72 meg., so it seems unlikely. But as mentioned above, you can monitor the size of oldSpace, and if it really is getting that big for good reason, you can use the VirtualMachineConfiguration class to increase that size. But note that any increase in oldSpace should be equally reflected by a change in arenaBytes. The arena is the total virtual space allocated, and oldSpace is carved out of that.

4) Can I turn off garbage collection?
The flipper can not be turned off, but it is so fast that it should not be of concern. Since compacts are controlled by the image, they can be turned off. The ProcessScheduler method #rebalance has a simple test, if oldSpace has grown by an amount larger than the oldSpaceThresholdIncrement, we need to compact old space. So to make sure no compacts take place, the oldSpaceThresholdIncrement could temporarily be assigned a very large number, and then set back at the appropriate time.

5. My application seems to have slowed suddenly, and garbage collection is not the problem.
It is possible for large applications to have a working code set that is bigger than our default code cache, which is 512K. Smalltalk is a dynamically compiled language. This means methods are compiled into a collection of byte codes, and the byte codes are compiled into machine code when the method is first executed and stored into the code cache. To test if the code cache is filling up, a bell can be added to ProcessScheduler>>codeCacheCleared (in version 3.0 or higher),
which is called every time the code cache is filled and must be cleared. If this is happening too frequently,
use the VirtualMachineConfiguration class to increase the size of the code cache.

Examples of setting VM values.
To change the nominalFlipSize permanently, including when the image is restarted, execute...
```
VM := VirtualMachineConfiguration fromImage. "Load current defaults"
VM nominalFlipSizeBytes: 65536.  "Set it to 64K"
VM toImage.  "Write it to the image"
```

To change the nominalFlipSize temporarily, execute..
```
VirtualMachineLibrary nominalFlipSizeBytes: 65536.
```

Note some VM settings will only change after restarting the image, like the size of newSpace.