Technotes

Download  Acrobat file (K)	Download  AppleWorks file (48K)

Strategies for Dealing with Low-Memory Conditions

Technote 1042 **May 1996**

One of the constants in writing software for the Macintosh is that sooner or later your application will run out of available memory. While applications running on other operating systems may use memory allocated from a global pool (backed by virtual memory), and can thus draw on an apparently limitless supply, each Mac application must work to avoid exhausting the fixed amount of memory the Process Manager gave it. Since Toolbox calls share the same heap with application calls, and parts of the Toolbox are notoriously poor at dealing with failed allocations, it's essential that you put into place a robust strategy to manage the problem. Applications that fail under low-memory conditions are not very friendly.

This Technote describes various strategies your application might take in dealing with low-memory conditions. It is also useful as a memory-management primer for those developers starting out on the Macintosh, or porting applications from other platforms.

This Technote expands on the chapter "Introduction to Memory Management" in *Inside Macintosh:Memory* . You should read at least pages 1-37 through 1-49 as a background to this Technote. Also, information on 68K segmentation assumes that you've read chapter 7 of *Inside Macintosh:Processes.*

Contents

- Maintaining a Memory Reserve
- Checking for Memory Allocations
- Informing the User of Low-Memory Conditions
- Using Temporary Memory
- Dealing With 68K Code Segments
- Special Problems with malloc and new
- Summary

Maintaining a Memory Reserve

The primary strategy for dealing with low-memory conditions is to maintain a memory reserve. Out of this reserve you can supply sufficient memory for small to moderate-sized allocations, as well as provide information to the rest of your application. By checking your reserve, you can determine the status of your memory supply. As the reserve supply dwindles, your app can gracefully restrict what the user can do by limiting her choices, in addition to informing the user that she's running out of memory.

Using A Purgeable Cushion

The advantage of using a purgeable cushion technique lies in its simplicity: there's little code to write. The disadvantage is that it can be difficult to ensure total safety when large portions of your application's memory is allocated indirectly by the Toolbox.

To create the memory reserve, allocate a handle early in your app's startup code. This handle should be in the range of 32K to 64K bytes in size, and should be marked as purgeable. When the Memory Manager later needs to create more free space, it will automatically purge any purgeable handles or resources.

During the main event loop, the application can check to see if the handle has been purged (if its master pointer is zero, it's been purged), and attempt to reallocate it with ReallocateHandle back to the original size. If this fails, then there is less than that amount of contiguous memory remaining, and the application can set a global flag to indicate that there is not enough memory to continue running as usual. It could then respond by warning the user to close documents, disabling menus that lead to memory-hungry functionality or in general keeping the user from pushing the limits of memory. When the handle can be reallocated again, the application can clear the global flag and reverse the steps it took earlier.

A useful technique is to only clear the global flag if you can recover the original memory plus an extra amount, as illustrated in the following code snippet:

```
extern Handle gReserveHandle;
if (*gReserveHandle == nil)
        {
        ReallocateHandle(gReserveHandle, kReserveSize + kSlopFactor);
        if (MemError()!= noErr)
                {
                SetHandleSize(gReserveHandle, kReserveSize);
                HPurge(gReserveHandle);
                SetReserveWasRecoveredFlag();
                }
        }
```

If the application is running right at the edge of the low-memory condition, using a "slop factor" can keep the warnings to the user from reappearing too frequently as the application "teeters on the edge."

How large to make the handle depends on your application's needs. If large Toolbox data structures, such as GWorlds or pictures, can routinely consume all available memory, then a size larger than 64K may be necessary. The only way to know is to experiment with different sizes and test your application's features near the limits of memory. If it dies, increase the reserve size; if it survives gracefully, you may try to lower the size.

Installing A Grow Zone Procedure

A slightly more complicated technique is to install a grow zone procedure using SetGrowZone. This procedure is called by the Memory Manager after it has exhausted all other strategies for finding free space, including purging purgeable handles and resources and compacting the heap. The grow zone procedure is called as the last resort. Often the grow zone procedure will have no idea whose allocation triggered the problem: it could be application code which will handle out-of-memory errors gracefully or Toolbox routines that fail to do any checking at all.

Grow Zone Proc Strategies

The grow zone procedure can attempt to provide additional memory by simply freeing or marking purgeable a single "cushion" memory block and allowing the main event loop to attempt to reallocate it, as previously described. The grow zone proc could also shrink the cushion handle with SetHandleSize and attempt to grow it in the main event loop.

The grow zone procedure can also undertake more complex and intelligent strategies to make memory available than the simple cushion techniques. The application may use buffers that can be shrunk or eliminated, or free up data structures stored in handles or pointers, or even mark additional resources as purgeable. It can also reduce its memory requirements in a gradual manner. Because the application may have knowledge of what blocks of memory and resources represent, it can free, shrink, or mark purgeable buffers to make memory available that cannot be made available automatically by the Memory Manager.

If the grow zone procedure returns with an indication that some memory was freed, the Memory Manager will again attempt another cycle of purging and compaction. It will continue calling the Grow

Zone proc and purging and compacting until either memory becomes available or the Grow Zone proc returns zero. In the latter case, the Memory Manager will give up and return `memFullErr' to the code which attempted the allocation.

During the main event loop, attempts should be made to recover the memory. As the reserve is used and then recovered, a global variable or state could keep track of what level of reserves remain, and make this information available to the rest of the application.

A List of Safe Memory Manager Calls

The following are the only Memory Manager calls that you can safely call inside the grow zone procedure:

- SetHandleSize, but only to a smaller size
- SetPtrSize, but only to a smaller size
- DisposeHandle
- DisposePtr
- EmptyHandle
- HPurge or the equivalent HSetState
- HUnlock or the equivalent HSetState

In 68K applications, you can also call UnloadSeg on segments not in the current call chain. See also the section "Dealing With 68K Segments" later in this Technote.

GZSaveHnd

You need to always call GZSaveHnd, which returns the handle on which the Memory Manager is currently operating, and not make any of the above calls with this handle.

What A Grow Zone Procedure Should Not Do

Basically, you can't make any calls that may directly or indirectly move memory or trigger additional requests for more free memory. These may result in recursive calls to the Memory Manager and to the grow zone procedure. The Memory Manager is *not* reentrant. Forcing it to rearrange the heap while it is already rearranging the heap is likely to cause unpredictable results, all of them unpleasant.

Examples of Things to Avoid in Grow Zone Procedures

Bearing in mind the caveat mentioned in the previous paragraph, here are some examples of things to avoid doing in grow zone procedures. Many of these problems originally came from developer questions over the years.

Don't allocate memory

This seems rather obvious but can be triggered indirectly by a variety of things. Most of the following problems are ultimately caused by allocating memory or causing the heap to be rearranged. You can never be completely sure that sufficient memory is available for even a small block. It's better to not risk confusing the Memory Manager.

Don't do synchronous file I/O

File Manager calls are only guaranteed to not move memory when called asynchronously. Unfortunately, doing asynchronous file I/O won't help the immediate need for releasing memory, so it isn't possible to make more room available by saving data to disk.

Don't update resources

Calls to ChangedResource and many other Resource Manager calls should be avoided because the Resource Manager uses the Memory Manager without taking into account the possibility that it might be called from the grow zone.

Don't put up dialogs

If you need to inform the user that memory is low, you should set a global flag and check it in your main event loop. Calling any user interface code during GrowZone is likely to allocate memory. Allowing the user to save documents from inside the grow zone procedure has been attempted, but without much success. Don't even think about it.

On 68K Macintoshes, don't call any routine that may force a segment to be loaded

This one can occur in very subtle ways. For example, in C++ you may decide to delete some objects that contain member variables that are Memory Manager heap handles or pointers, and thus free up some space. You call *delete* on the object. Unfortunately the destructor for the object's class, or a destructor for a superclass, resides in a code segment that is currently not in memory. Calling this code to free up some space actually consumes more space, leading to recursive calls to your grow zone procedure and probable crashes. Make sure everything that can be called from your grow zone procedure is kept in resident segments.

Checking for Memory Allocations

Even with a low-memory strategy in place, it is still very important to always determine if memory allocations have succeeded. However, careful use of purgeable cushions or grow zone procedures can greatly simplify the handling of failed allocations. If you have a global flag or function which returns the current state of the cushion or memory reserve, you can quickly check to see if you have sufficient memory to perform a task or operation in your application. If the reserves are low, you can abort the task at the start. If the reserves are intact, then there is at least that much memory available, and you can proceed with confidence.

You can avoid checking some allocations for failure, as long as you can be certain your memory state indicates the reserve or cushion will supply whatever memory is needed. You should avoid making potentially time-consuming calls such as PurgeSpace to determine if enough memory is available. This type of call could be saved until you need to check a memory requirement larger than could be supplied by the reserve or cushion.

Informing the User of Low-Memory Conditions

Letting the user know that memory is running low is a delicate art. If the application is running with very little memory, it may be possible to get into a state where the "out of memory" alert arrives continuously, which is highly annoying.

Developing a Two-Stage Plan

A good thing to do is to wait a fixed amount of time between warnings in order to allow the user time to quit, close documents, or perform other actions. It is also useful to have a two-stage plan:

1. A warning that memory is becoming low (which could be shown when the memory reserve is shrunk, then reallocated)

2. A more serious indication that there is no memory remaining (when the memory reserve cannot be recovered)

At the second stage, most program functions should be made unavailable, except for those involved in closing documents, saving, etc. It's always better to keep the user from reaching the absolute limit than to sadly inform her that "the application is out of memory, and will have to quit now," or even greet her with a rude bomb alert.

Important:
Since memory alerts appear when there isn't very much memory left to work with, the resources and code needed to display them should be resident in memory. Many applications collect alert strings into a single STR# resource. If the Out of Memory message is in such a large resource, it may not be possible to load the resource to extract the individual string. It's better to keep these alerts resident in memory as much as possible with the string as a static item in the alert.

Guaranteeing the User Can Save Documents

You always want to guarantee that the user can save documents when the application is running with low memory. This seems fairly obvious, but saving documents can use varying amounts of memory to convert in-memory data to a file format, or save resources, such as edition records or preview images. It may be necessary to keep a separate reserve of memory strictly for saving documents.

You may also want to inform the user when a low memory crisis has been averted, i.e., now the user can relax because she's closed the necessary documents.

Note:

Applications that show memory availability in their About box should ensure that opening the About box when memory is very low does not exhaust what little memory remains.

Using Temporary Memory

Temporary memory allocations can assist your application if it is running low on memory. If the app detects that memory is low, it can check to see if temporary memory is available, and switch some allocations to using TempNewHandle. However, it should be stressed that this is only useful for items of a temporary nature, not data that will exist throughout the lifetime of the application.

When you allocate memory in the temporary heap (actually the Process Manager heap), you may make it difficult for the user to launch additional applications, since the applications will be launched into this heap. Your application also can't count on having any temporary memory available, so alternative strategies must always be designed for dealing with its absence.

Dealing With 68K Code Segments

Applications that continue to support the Motorola 68000 family have the additional burden of managing code segments. If the Segment Loader can't load a segment (usually due to insufficient memory), you'll get System Error #15. Another problem you may have is limiting the amount of memory used for code segments to the minimum amount necessary at any one time.

Here are a few good strategies for dealing with 68K code segments.

A Few Good Strategies

All strategies begin by dividing code into "resident" and "non-resident" segments. Resident segments contain the main event loop, the grow zone procedure, any interrupt time code, and any other code frequently referenced. The remaining segments should be based on an analysis of which sections of code get called together.

The goal is to minimize the number of segments (and memory) needed for each operation or feature in the application. To accomplish this, you need to understand which parts of your app get called for each operation. A utility program such as Metrowerks' ZoneRanger can let you watch where the memory segments reside as your application executes.

Use a Single Segment

The simplest strategy is to build your application with an option like MPW's model far and use a single segment. This way everything is in memory, and there is no code to write. If you can afford the memory footprint, it makes no sense to complicate memory management with segmentation. (In single-segment CFM applications, having all the code loaded all the time is the only option you get anyway, although virtual memory can help.)

This may be ideal if your application is small, or memory requirements are not an issue -- e.g., for some vertical applications. Also, there's another important issue with regard to single-segment applications. If the application is going to be used on a PowerBook computer, then having a single segment is very desirable -- multiple segments will keep the hard disk spinning and decrease battery life.

Unload Segments in the Event Loop

The next simplest strategy is to call UnloadSeg with a routine from each non-resident segment each time through the event loop (after the event has been handled, or right before WaitNextEvent). If your

application uses little memory beyond what is needed by the code, and especially if it makes only small permanent allocations from one call to WaitNextEvent to the next, then this may be sufficient by itself. Assuming that the division of code into segments was well done, the only problem is to determine a routine to use in calling UnloadSeg for each segment. Here are two examples:

1. Create a special routine in each segment with the name of the segment, and use that routine address in UnloadSeg. The segment contents can then be shifted around, without the need to search for a new routine to use, or worry if you've missed something.

2. Get the address directly by scanning the jump table for the first routine in each segment. This is highly dependent on the jump table format produced by your development environment. See the MacApp source file "USegments.cp" for an example.

Unload Segments at Any Time

At any point during the application's execution, you could conceivably unload any code segments not currently in the call chain.

The way to determine this at runtime is to scan the stack for any addresses contained within loaded segments. The steps to take each time are:

1. Construct a table with entries for each segment, containing the current memory location and size for each segment in memory, and a `keep' flag. Mark all resident segments' flag with true; mark all others as false.

2. Starting at the current value of register A7, scan to the address returned by LMGetCurStackBase, skipping two bytes at a time. Consider each value as a pointer. If this value is contained within the application heap zone, check to see if it is contained within a segment (using the table you constructed in Step #1). If it is, set the keep flag to true for that segment. Here you are basically looking for return addresses put on the stack during subroutine calls, which would indicate code in the call chain.

3. (using the technique of your choice to obtain an address from the segment).

A somewhat complex example of this can also be found in the MacApp source file "USegments.cp".

Patch LoadSeg

Another alternative solution is to patch LoadSeg. The patch will check to see if sufficient memory exists to load the segment from disk (if it has been purged). (One way to do this is to simply load the segment resource with a call to GetResource.) If the patch finds room, it then calls the real LoadSeg. If there isn't enough memory, the patch tries to make room available by performing the actions of a grow zone procedure such as releasing buffers, shrinking reserves, or unloading code segments not currently in use. Then the patch goes ahead and calls the real LoadSeg. This way a segment load will always succeed (provided the underlying reserve memory and segmentation scheme is sound).

For cases in which a segment cannot be loaded, you might want to throw a (C++) exception.

Special Problems with malloc and new

Applications written in C++, and applications using the standard C allocators (particularly those ported from other platforms) have special problems in mixing their memory allocations with the Memory Manager.

All C/C++ development environments provide versions of malloc and new which sub-allocate out of Memory Manager blocks created with NewPtr. These blocks are non-relocatable, and thus can severely fragment the heap, especially when memory is becoming scarce.

An additional problem is that some of these implementations of new and malloc fail to release these NewPtr blocks when they no longer contain any sub-allocated information. As these blocks begin to fill up the heap, there is less and less space available for Toolbox items such as windows, menus, or resources. It is important to limit the growth of these blocks to keep sufficient room available for Memory Manager allocations.

Some implementations of malloc and new allow you some control over their behavior; check your compiler's documentation for details. There are also commercially available alternative allocators, such as Smartheap, which may be of some help in dealing with this problem. Another possible solution is to create alternative heap zones within the application zone and keep all non-Memory Manager allocations within them.

Summary

Dealing with low-memory conditions is necessary for successful Macintosh application development. This Note outlines a number of strategies that you can follow in order to handle low-memory conditions. Maintaining a memory reserve, for example, is essential to keep your application from dying when it runs out of memory. Other strategies, such managing 68K code segments, or properly checking for failed memory allocations, are also important in building quality applications for the Macintosh. Failing to devise a proper memory strategy may result in a product that is badly behaved and frustrating to your customers.

- *Inside Macintosh: Memory*
- *Inside Macintosh: Processes*
- MacApp 3.3 on ETO #19

'
