ME 14 - The New Memory Manager and You

Memory

Written by: Brian Topping December 1993

This Technical Note describes changes in the Modern Memory Manager that you need to be aware of. Specifically take note of the changes to the bus error handlers in the first section.

Topics

Introduction

The introduction of the PowerPC Macintosh also introduces a new Memory Manager. Many of the splendid features of this new implementation have already been discussed in tech note Memory 13. Weary travelers may have noted some strange behavior though, and this note attempts to answer most of the remaining big questions you may have about it.

Bus Error Handlers

When 32-bit QuickDraw was introduced around the Macintosh IIci, it became immediately possible that frame buffers would exceed the maximum address space available within a Macintosh NuBus slot with 24-bit addressing. (As you may remember, NuBus slots have only 1/16th of the addressable space when accessed in 24-bit mode). This problem was initially solved by making a 32-bit addressing mode that could be enabled via `_SwapMMUMode`. The swap would allow the entire frame buffer to be accessed, but would also cause the 24-bit Memory Manager and application clients to crash when they tried to use 24-bit clean master pointers. In order to minimize MMU swapping, bus error handlers were installed in the Memory Manager to catch rouge addresses that may be propagated by QuickDraw. If a 24-bit clean handle was passed by QuickDraw to the Memory Manager when the MMU was swapped to 32-bit mode, the ensuing bus error would fire the bus error handler, which would run the address thru `_SwapMMUMode` and try again, failing only after trying once.

When the 32-bit clean Memory Manager was introduced, a (then helpful) side effect of the bus error handlers was to filter bad addresses and gracefully return an error code instead of crashing. Unfortunately, bus error handlers are still there watching out for 24-bit handles. Programmers may not be passing in bad handles any more, but the bus error handlers watch to make sure that all the major dereferences to find the zone do not fail. Because the handlers were there protecting you from yourself (much like the USA does with seat belt laws), you didn't notice some faulty address calculations in your code actually cause a bus error in the Memory Manager. All your code would have seen is a -111, `memWZErr`, or -113, `memAZErr`. If you weren't checking these (for instance after an `_HLock`), your code would have made it through Quality Assurance and you would have shipped.

Commonly, these faulty calculations come from assuming the size and shapes of block and heap headers. This is a bad thing. The sizes of block and heap headers have both changed with the Modern Memory Manager, and are bound to change again in the future. Don't rely on undocumented features, including features such as structure sizes that you deduced from the documentation but were not explicitly documented. If you have concerns whether you will break in the future, it is best to contact DTS now.

What Was Done

Unfortunately, the bus error protection provided by the Memory Manager was very time consuming, and with the introduction of future operating systems that protect these vectors, it was not going to get any faster. Most importantly, now that addresses are always real logical addresses, the handlers are just extra baggage. The best option was to completely remove the handlers, and this was tried unsuccessfully in early versions of the Modern Memory Manager shipping with PowerPC. Unfortunately, there are still too many important clients that rely on the side effects provided to simply remove them.

Out of the ashes of this came a happy medium. On the PowerPC machines with the Modern Memory Manager, bus error handlers are installed, but the first stop on the exception journey is to the PowerPC debugger, not directly to the Memory Manager exception handler. Should you be one of the lucky abusers of the Memory Manager, you will see the debugger stop in the Memory Manager code with an 'Access Fault'. Unlike using EvenBetterBusError though, you are able to recover from this. In the 'Control' menu of the PowerPC debugger, there is a 'Propagate Exception' item. If the debugger host is not connected to the nub, the nub will eventually time out and propagate the exception on it's own, resulting in an apparent 'freeze' every so often as the nub times out before passing on the exception.

Remember when you propagate the exception, your chance to debug the problem is lost. Fixing these problems in your own source early on is the best way to feel confident that late in testing you will not have to propagate exceptions, only to find that it wasn't the Memory Manager, and that there was no exception handler to catch your fall.

There are some lessons to be learned here. The first is if you administrate systems as a hobby or know of someone who does, be sure that they do not put the PowerPC Debugger Nub on end-user PowerPC machines. Users will be confused as their machines freeze for 15-30 seconds at a time, then suddenly continue at the blazing speed PowerPC's are known and loved for. The second lesson is that if you are developing on a PowerPC and you see an access fault, try to track down who it is. If it is in your code, fix it. If it is in someone else's code, try to contact them and get them to fix it. If you like the set of Extensions, Control Panels, and Applications you are currently using and they cause access faults, you will be mighty unhappy when the safety net is removed and they don't all work in the next release. Your letter or phone call to the developer might make the difference. And if it is your code, it is pure suicide not to test at this point on a PowerPC.

The bus error handlers, even though they are implemented better than their ancestors, are not cheap. They *will* be removed at the next speed release.

Bus Error Return Values

On the subject of the bus error handlers, the Modern Memory Manager no longer returns both -111, `memWZErr` and -113, `memAZErr`. Only -111, `memWZErr` is returned, no matter whether the bus exception occurred in the dereference of a zone or memory address.

The difference between these errors was the byproduct of the need to know which address needed to be passed to `_StripAddress`. Depending on the error, either the heap or memory reference would be stripped, and the entire operation would be repeated. Since the Modern Memory Manager only works on 32-bit clean systems, this is not an issue. While the old behavior could have been implemented, the cost to install separate handlers as different code paths are entered is prohibitive. As such, the more commonly returned error, -111, `memWZErr` is always returned when a bus error happens in the Memory Manager, at least until the next release when the handlers are removed and the system will simply crash.

Free Block Miscellanea

While on the subject of the new Memory Manager, it is important to remind you that disposing blocks is hazardous to their integrity. Random data scattered throughout the newly freed block are targets for our gratuitous and spiteful clobbering. This has been documented before but it never hurts to mention it again.

While this point was undoubtedly well taken, only top scorers on KON & BAL's Puzzle Page figured out that there are other side effects that will bring this behavior to life. While this is no attempt to list all of them, the general ideas here will be enough to seed your imagination, and maybe even get a better score on the next Puzzle Page.

The first is closing out a resource file while the data in the resources belonging to that file are still in use. Inside Mac tells you that `_CloseResFile`, among other things, walks the resource map and "for each resource in the resource file, releases the memory it occupies by calling the `_ReleaseResource` procedure". `_ReleaseResource` of course calls `_DisposeHandle`. We all know what `_DisposeHandle` does to your data by now; data integrity was not in the list. The moral is that you shouldn't close out a resource file until you are done with all of the resources that were contained in it, unless you explicitly call `_DetachResource` on each resource you intend to keep using before you close the file.

A similar situation occurs when there is a purgeable handle (including purgeable resources) around and you expect to use it after calling something that allocates memory. This has always been a problem, but can be a problem in different ways now that the integrity of freed blocks is guaranteed nil and the dynamics of the memory manager are different, including different algorithms and different block sizes causing the different algorithms to act differently (see, things really are different). One bug we saw recently was that `_AddResource` was being called on a purgeable handle. If `_AddResource` decides that the resource map for the file needs to be grown, the heap may well be compacted, thereby purging the resource. This will clobber the data in the block, causing the data that was added to be mangled.

You can find these pretty easily though. Using the ZapHandles extension, you would immediately fail in any attempt to use a disposed block because the `_DisposeHandle` and

`_DisposePtr` get head patched to clobber the data in the block. No waiting around with this extension, your program will crash very soon after the block is used again. Other problems with purgeable blocks, among other things, can be found with Heap Scramble, a feature of your favorite debugger. Be sure to get a debugger that is compatible with the new Memory Manager; old versions are a shining example of why not to rely on the structure of blocks (they have an excuse though). This will tend to blast away blocks that don't have a permanent home. Shipping a product without trying Heap Scramble first usually ends up as an exercise in embarrassment.

A5 World Problems and Heap Callback Procedures

Many Memory Manager savvy applications use Grow Zone and Purge procedures to indicate when they should clean up or release memory so that the system can continue to function. A common technique is to register a grow zone procedure (using `_SetGrowZone` or the like) that is called upon to release or resize smaller a block of memory allocated when memory is plentiful, often at application launch. Applications commonly track space that can be freed in global pointers or structures that are accessible through the A5 world. As such, the A5 world must be set up by the Memory Manager before the grow zone procedure is called. This is commonly done by calling `_CurrentA5`, and setting the 680x0 register accordingly.

Unfortunately, `_CurrentA5` may not always reflect the correct A5 value for the heap that is being operated on. This is true in rare cases such as an application in the background that is both critically low on memory and whose update or visible regions need to be changed to reflect changes that the foreground applications are making to the screen. Because updates to the screen do not actually cause the Process Manager to do a minor switch, `_CurrentA5` will not be properly set. If the region of the background application that is critically low on memory needs to grow <u>and</u> the application has a grow zone procedure registered which relies on the parent applications A5 world <u>and</u> the grow zone needs to be called to free up memory, the grow zone procedure will be called with the frontmost application's A5 value, since it comes from `_CurrentA5`. This will of course cause all hell to break loose as the grow zone procedure tries to use data it thought was at a specific offset off A5, when in fact that data is some other application's different use of the offset.

Since this problem is so rare, we did not come across it until just recently. Options for those truly stuck by this or not interested in finding this is a problem later include tracking your own A5 via `_Gestalt` or some other global registration scheme.

Conclusion

With the exception of the Heap Callback problems, the gray area of where programs work but are not really correct in the eyes of the Memory Manager is getting smaller. This may seem like a burden, but all of these points are problems waiting to happen, whether you are using the Traditional or Modern Memory Manager. Correcting them now will ensure that as the rules get stricter, you are not left in the compatibility doghouse with your customers.

Further Reference:

- *Inside Macintosh*, Designing Cards and Drivers for Macintosh
- Technical Note ME 13 - Memory Manager Compatibility