Technote 1122

Locking and Unlocking Handles

CONTENTS

This Technote is addressed to developers who manipulate Memory Manager handles, and in particular those who lock and unlock them. It explains why and when handles need to be locked and how and when to preserve the state of a handle to avoid erroneously unlocking it.

After the discovery of a subtle bug (in both Apple and third party code) having to do with the erroneous unlocking of handles, DTS recently conducted an exhaustive search of developer documentation for an unequivocal and complete statement about proper methods for locking and unlocking handles. Since no such documentation could be found, this Technote now serves as that statement.

Change history:

1.0	02/09/98	initial version
1.0.1	02/13/98	clarified examples of system calls which move memory

The Basics

A handle refers to one kind of memory block created and maintained by the Memory Manager. Because a handle is an indirect reference (namely, a pointer to a pointer) to a block, the Memory Manager is free to move such blocks at certain well-defined times. The pointer to which a handle points is called a master pointer. When a handle block moves, the Memory Manager changes the master pointer, not the handle. (For a more detailed review, check the Heap Management section of the Introduction to Memory Management chapter of *Inside Macintosh: Memory*).

There are times at which a handle needs to be locked. If a program needs to pass a master pointer (or a pointer to bytes inside a handle block) to a routine which can move memory (that is, a routine which calls the Memory Manager, directly or indirectly, in a way which might cause memory to move), the handle must be locked. A simple example is the first parameter to `PtrAndHand`. A slightly more complex example is any call to `NewWindow`. Finally, a subtle example of a routine which moves memory is one whose implementation does not move memory but whose mere invocation does. Calling a routine which resides in an unloaded segment resource may cause handle blocks to move when the segment loads. (For details, see the Loading Code Segments section of the Introduction to Memory Management chapter of *Inside Macintosh: Memory* and the Segment Manager chapter of *Inside Macintosh: Processes* .)

Since leaving a handle locked indefinitely may prevent the Memory Manager from relocating other blocks optimally, it is desirable to unlock a handle when it is no longer necessary for the handle to be locked. However, it's dangerous to unlock a handle without understanding all the implications, even in many

simple cases. It's more dangerous to unlock a handle your program does not own.

The Problem

The problem is that the Memory Manager does not track the number of times a handle has been locked. A handle is either locked or unlocked. If a program locks a handle twice and then unlocks it once, the handle is unlocked. This means that non-trivial programs cannot simply balance lock and unlock operations. Here's an example of how **not** to maintain a lock on a handle:

```
static pascal OSErr Unsafe_DoSomethingToHandle (Handle h)
{
    OSErr err = noErr;

    HLock (h);

    if (!(err = MemError ( )))
    {
        err = DoSomethingToPointer (*h);

        HUnlock (h); // Danger, Will Robinson!

        if (!err)
            err = MemError ( );
    }

    return err;
}
```

If the caller of `Unsafe_DoSomethingToHandle` has locked the parameter handle and expects it to stay locked, the program is in for a nasty surprise. Even though the caller has locked the handle it passes to `Unsafe_DoSomethingToHandle`, the call to `HUnlock` in `Unsafe_DoSomethingToHandle` will undo the lock.

There are, of course, cases which are more difficult to debug. Sometimes the software that needs the handle to stay locked is not the direct caller, but another function the caller calls, or even part of the system. For an example of this last case, see Technote 1118, "Unlocking GDHandles Considered Harmful".

The Solution

The general solution is to make sure the handle state is preserved by the function (or code sequence within a function).

> **Important:**
>
> If you always preserve the handle state, you will always be safe. There are times when preserving the handle state may be unnecessary and sub-optimal. However, it's essential to precisely understand if and why a given case is safe before you choose not to preserve a handle's state. (More about these exceptional cases later. If you are a defensive programmer and would prefer to be in the habit of writing safe code every time, feel free to ignore the exceptional cases.)

The handle state is a collection of bit flags maintained for each handle by the Memory Manager. One of the state flags describes whether the handle is locked. Programs can obtain the state of a handle by calling `HGetState`. All of the techniques listed below use `HGetState` in some way.

Technique #1

The simplest technique is to save the entire handle state, lock the handle, and then restore the entire handle state. This means that the bit which represents whether the handle is locked has the same value before and

after the code runs, so if the handle is locked before the code runs, it stays locked, and if the handle is unlocked before the code runs, it stays unlocked. Here's a typical sequence of calls:

```
static pascal OSErr SafeOne_DoSomethingToHandle (Handle h)
{
    OSErr err = noErr;

    SInt8 hState = HGetState (h);

    if (!(err = MemError ( )))
    {
        HLock (h);

        if (!(err = MemError ( )))
        {
            err = DoSomethingToLockedHandle (h);

            HSetState (h,hState);

            if (!err)
                err = MemError ( );
        }
    }

    return err;
}
```

This technique is reasonably safe. The disadvantage is that it saves and restores the **entire** handle state. If the function `DoSomethingToLockedHandle` changes any of the other handle state flags (for example, the flag which controls the handle's purgeability), the call to `HSetState` will wipe out the change.

Technique #2

A slightly more complicated technique involves testing the handle state to see if the handle is locked and then locking and unlocking it only if needed. Here's a typical sequence of calls:

```
enum { kHandleLockMask = 0x80 }; // absent from <MacMemory.h> 3.0.1

static pascal OSErr SafeTwo_DoSomethingToHandle (Handle h)
{
    OSErr err = noErr;

    SInt8 hState = HGetState (h);

    if (!(err = MemError ( )))
    {
        Boolean handleWasLocked =
              (kHandleLockMask & hState) ? true : false;

        if (!handleWasLocked)
        {
            HLock (h);
            err = MemError ( );
        }

        if (!err)
        {
            err = DoSomethingToLockedHandle (h);

            if (!handleWasLocked)
            {
                HUnlock (h);

                if (!err)
```

```
                    err = MemError ( );
                }
            }
        }

        return err;
    }
```

This sequence avoids the problem that a stale handle state might be restored following the call to `DoSomethingToLockedHandle`. The disadvantage here is the relative complexity. This is the kind of code that nobody likes to think carefully about (well, perhaps nobody but DTS geeks), and if you must lock and unlock a lot of different handles, it's likely that bugs will creep in somewhere.

One way to avoid this kind of risky tedium is to wrap the logic up in a C++ class whose constructor locks the handle (if necessary) and whose destructor unlocks the handle (if necessary). This extra layer of wrapping on Technique #2 even relieves you of having to write code for error-handling and exceptions, since the handle will be unlocked, if appropriate, whenever the lock object falls out of scope.

Exceptional Cases

There are, of course, cases in which you can avoid saving and restoring a handle's state. You may, during the optimization phase of your development, want to look for cases in which you can do this, to avoid code bloat and if you find you are making "too many" (whatever that means for your program) Memory Manager calls in a tight loop. Be sure to measure before assuming you must optimize. Also: be safe, then fast.

Don't Lock the Handle

Sometimes you can avoid saving the handle state by not locking a handle. This seems like an obvious statement, but, many times, code locks handles when it doesn't need to. Make sure you understand just when the Memory Manager can move relocatable blocks (perhaps implicitly via some other Manager which calls Memory Manager). For example, DTS commonly sees developer code which looks something like this:

```
    static pascal OSErr UnnecessaryHandleLock (Handle h)
    {
        OSErr err = noErr;

        SInt8 hState = HGetState (h);

        if (!(err = MemError ( )))
        {
            HLock (h);

            if (!(err = MemError ( )))
            {
                **h = 12; // we locked 'h' for this?

                HSetState (h,hState);
                err = MemError ( );
            }
        }

        return err;
    }
```

In addition to taking advantage of situations which don't require you to lock a handle, you can also deliberately go out of your way to avoid locking it. If you need to modify a small portion of a handle block, you can copy the appropriate bytes into a variable, modify the variable, and copy the variable back into place within the handle. For example:

```
    static pascal OSErr AvoidHandleLock (Handle h)
```

```
{
    OSErr err = noErr;

    char c = **h;

    err = DoSomethingToCharacter (&c);

    if (!err)
        **h = c;

    return err;
}
```

Exclusive Access

If a well-defined part of your code has exclusive access to a handle, that code can lock and unlock handles at will.

Perhaps you have a code module which is responsible for managing some handle-based buffers which are "hidden" from the module's callers inside an opaque data type. In this case, your code allocated the handles and is solely responsible for them throughout their entire lifetime, thus your code can manipulate those handles any way it likes.

Another example might be a function which needs to allocate a temporary buffer (and, to avoid the obvious memory leak, dispose the buffer as the function exits). Since it's safe to pass a locked handle to `DisposeHandle`, even though `NewHandle` produces an unlocked handle, a code sequence like this one is perfectly acceptable:

```
static pascal OSErr SafeThree_DoSomethingToHandle (void)
{
    OSErr err = noErr;

    Handle h = NewHandle (12);

    if (!h)
        err = MemError ( );
    else
    {
        HLock (h);
        err = MemError ( );

        if (!err)
            err = DoSomethingToLockedHandle (h);

        DisposeHandle (h);

        if (!err)
            err = MemError ( );
    }

    return err;
}
```

Summary

It's generally not safe to preserve a handle's lock state simply by balancing calls to HLock and HUnlock. In most cases, you should instead preserve the handle state. In the remaining cases, you need to fully understand the implications of not preserving the handle state. In fact, if safety is of utmost importance to you, you can't go wrong by always using one of the handle state preservation techniques illustrated in this Technote.

Further References

- the Memory Manager chapter in *Inside Macintosh: Memory*
- the Segment Manager chapter of *Inside Macintosh: Processes*


Acrobat version of this Note (K)

Acknowledgements

Major funding provided by the Handle State Preservation Society.

To contact us, please use the Contact Us page.
Updated: 15-April-98