Technical Note TN2059
Using collection classes safely in multithreaded applications

CONTENTS

This technote describes some problems that can occur when using mutable collection classes (arrays, dictionaries, and sets) in multithreaded Cocoa applications, and describes some approaches to solving these problems, including several implementations.

Multithreaded Core Foundation applications may involve the same concerns, since Cocoa and Core Foundation classes are "toll-free bridged." This note does not address Core Foundation collections explicitly, but most of the principles discussed here apply to Core Foundation as well. For more information on Core Foundation, see Overview of Core Foundation in the References section.

This note doesn't cover general issues in threading. For an introduction to those, see the Reference section for a list of readings.

[Sep 10 2002]

Problems with shared data in threaded applications

Whenever two threads share data, they must synchronize access to that data to avoid bugs that can arise when they both work with the data at the same time.

In most cases, a thread synchronizes by using a lock to prevent other threads from entering the same code at the same time. For example:

```
NSLock *statisticsLock; // assume this object exists
int statistics;
        ...
[statisticsLock lock];  // make sure no two threads...
statistics += 1;        // ...update 'statistics' at the same time
[statisticsLock unlock];        // release the lock
```

The use of `statisticsLock` above ensures that only one thread executes the "+=" operation at a time. Without this, two threads might try to update the same location at the same time, losing one thread's contribution to the statistics.

Back to top

Problems with collection classes in threaded applications

When your code works with a mutable dictionary or array, locking the object only during the time the code works with its contents may not prevent all bugs.

To see why, consider this code (all the code shown here uses dictionaries, but the same concerns apply to arrays and sets):

```
        NSLock *dictionaryLock; // assume this object exists
        NSDictionary *aDictionary; // assume this object exists

        NSString *theName;
                ...
        [dictionaryLock lock];
        theName = [aDictionary objectForKey: @"name"];
        [dictionaryLock unlock];

        NSLog (@"the name is '%@'", theName);
```

This code protects access to the mutable dictionary; however, even if all other parts of the program use the same lock to avoid concurrent use of the dictionary, the code above can fail. Here's how:

When a mutable collection removes an object, it sends the object a `release` message. Keeping this in mind, consider the following sequence:

1. `aDictionary` contains the name `@"Pat"` for the key `@"name"`, with a retain count of 1
2. thread "A" executes the code above: it locks, sends `objectForKey`:, and unlocks — the variable `theName` now has a value of `@"Pat"`, with a retain count of 1
3. thread "B" now locks the lock and updates the dictionary with:

   ```
   [aDictionary setObject: @"Sandy"  forKey: @"name"];
   ```

   The dictionary sends `release` to the old value for this key, the string @"Pat". Since the retain count was 1 to start, this release drops the count to zero, and the string `@"Pat"` gets deallocated.

4. thread "A" now tries to use its variable `theName` in the `NSLog(...)` invocation, but the variable refers to the string `@"Pat"` which has been deallocated, leading to a crash or other random results.

The above sequence typifies the subtlety of using collection classes in threaded applications, but is not the only concern you need to keep in mind. Let's briefly consider a second example.

Suppose your code adds an object to a dictionary. Since the dictionary retains its values, the code releases the object after adding it. The code looks like this:

```
NSString *theName; // assume this exists
        ...
[aDictionaryLock lock];
[aDictionary setObject: theName  forKey: KEY];
[theName release]; // since dictionary retains it, we don't need to
[aDictionaryLock unlock];

NSLog (@"the name is '%@'", theName);
```

The risk here resembles the risk in the previous code: Once the code unlocks, another thread may change the dictionary, releasing the value. Again, `theName` refers to a value that can get deallocated in the short period between unlocking and trying to use it in the `NSLog(...)`.

The same concerns apply to objects stored in arrays and sets, not just dictionaries: If thread B removes from an array or set an object to which thread A holds a pointer, that object may not remain valid during the scope in which thread A plans to use it.

To sum up:

- When multiple threads share a collection object, they need to synchronize access to that collection.
- Collection objects send release to objects as they remove (or replace) them.
- After one thread adds an object to a collection, or gets it from a collection, a second thread may cause that object to become invalid.
- Thus, when multiple threads share a collection object, they also share all values contained in the object, even when one thread holds a reference to the value.

Back to top

Example 1: A faulty application

The following source code illustrates the risks of sharing collections across threads. Even though it uses a lock to prevent simultaneous access to a mutable dictionary, it still crashes consistently.

```
#import <Foundation/Foundation.h>

static NSMutableDictionary      *aDictionary = nil;
static NSLock                           *aDictionaryLock = nil;

@implementation NSMutableDictionary (Churning)

#define KEY             @"key"

- (void) churnContents;
{
    unsigned long       i;

    for (i = 0; ; i++)
    {
        NSAutoreleasePool       *pool;

        pool = [[NSAutoreleasePool alloc] init];

        [aDictionaryLock lock];
        [self setObject: [NSString stringWithFormat: @"%d", i]  forKey: KEY];
        [aDictionaryLock unlock];

        [pool release];
    }
}

@end

#define COUNT   10000

static void doGets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
    {
        NSObject        *anObject;

        //      Get the dictionary's value, and then try to message the value.
        [aDictionaryLock lock];
        anObject = [aDictionary objectForKey: KEY];
        [aDictionaryLock unlock];

        [anObject description];
    }
}

static void doSets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
    {
        NSObject        *anObject;

        anObject = [[NSObject alloc] init];

        [aDictionaryLock lock];
        [aDictionary setObject: anObject  forKey: KEY];
        [anObject release];
        [aDictionaryLock unlock];

        [anObject description];
    }
}

int main ()
{
    SEL         threadSelector;
```

```
    [[NSAutoreleasePool alloc] init];

    threadSelector = @selector(churnContents);

    aDictionary = [NSMutableDictionary dictionary];
    aDictionaryLock = [[NSLock alloc] init];

    //  Start the dictionary "churning", repeatedly replacing the
    //  sole value with a new one under the same key.
    [NSThread detachNewThreadSelector: threadSelector
                toTarget: aDictionary
                withObject: nil];

#if 1 // because this crashes, you can turn it off to show that doSets() also crashes
    doGets();
#endif
    doSets();

    return 0;
}
```

Listing 1. main1.m (available in the Downloads section)

Bugs in multithreaded applications may appear only sporadically. So that you don't have to run it repeatedly to see the problem, this application spawns a thread that sends the `churnContents` message to a mutable dictionary. This method repeatedly replaces one value in the dictionary with a sequence of objects.

It includes two functions `doGets()` and `doSets()`, either of which will crash the application — you can comment out the call to the first function to see the other one crash. The `doGets()` function repeatedly gets a value from the dictionary and sends it a description message, eventually causing a crash.

The `doSets()` function puts a value into a dictionary, releases the value on the assumption that the dictionary will retain it, and similarly risks a crash by sending `description` to the object. (The choice of `description` has no significance; any method would do the same thing.)

Back to top

Safeguarding shared objects

How can code safely work with objects it gets from a collection or stores in a collection? One approach is to send the object a `retain` message and later — when you're done using it — send it a balancing `release` message. The net effect of these two actions leaves the object unchanged, but during the interval between retaining and releasing your code lays claim to the object, preventing it from getting deallocated by other threads.

The problem with this approach is that you have to remember to send the `release` message, or else the object may never get deallocated, and "leak." Instead of waiting until you're done and sending `release`, you can immediately send it an `autorelease` message, which guarantees that the object will get released when the current autorelease pool gets deallocated.

Think of retain-and-release as asking someone to lend you a dollar and you promising to pay them back. Think of `retain-and-`**`autorelease`** as asking someone to lend you a dollar and them promising to make you pay it back. The latter approach is more conservative, in that it ensures your books stay balanced. (Keep in mind that autoreleasing takes more time and space than releasing, though.)

To make the code above safe, you need to add one line, noted in **boldface** below, to the code given before Example 1:

```
NSLock *dictionaryLock; // assume this object exists
NSDictionary *aDictionary; // assume this object exists

NSString *theName;
        ...
[dictionaryLock lock];
theName = [aDictionary objectForKey: @"name"];
[[theName retain] autorelease]; // keep object around for now
[dictionaryLock unlock];

NSLog (@"the name is '%@'", theName);
```

The combined `retain` and `autorelease` may look like it has no effect, but the `retain` takes effect immediately, while the `autorelease` undoes the `retain` later. In the end, the two messages cancel out, but the `retain` keeps the object from getting deallocated while your code is using it.

Back to top

Example 2: A corrected application

```
#import <Foundation/Foundation.h>

static NSMutableDictionary      *aDictionary = nil;
static NSLock                           *aDictionaryLock = nil;

@implementation NSMutableDictionary (Churning)

#define KEY             @"key"

- (void) churnContents;
{
    unsigned long       i;

    for (i = 0; ; i++)
    {
        NSAutoreleasePool       *pool;

        pool = [[NSAutoreleasePool alloc] init];

        [aDictionaryLock lock];
        [self setObject: [NSString stringWithFormat: @"%d", i]  forKey: KEY];
        [aDictionaryLock unlock];

        [pool release];
    }
}

@end

#define COUNT   10000

static void doGets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
    {
        NSObject        *anObject;

        //      Get the dictionary's value, and then try to message the value.
        [aDictionaryLock lock];
        anObject = [aDictionary objectForKey: KEY];
        [[anObject retain] autorelease];
        [aDictionaryLock unlock];

        [anObject description];
    }
}

static void doSets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
    {
        NSObject        *anObject;

        anObject = [[NSObject alloc] init];

        [aDictionaryLock lock];
        [aDictionary setObject: anObject  forKey: KEY];
```

```
        [anObject autorelease];
        [aDictionaryLock unlock];

        [anObject description];
    }
}

int main ()
{
    SEL         threadSelector;

    [[NSAutoreleasePool alloc] init];

    threadSelector = @selector(churnContents);

    aDictionary = [NSMutableDictionary dictionary];
    aDictionaryLock = [[NSLock alloc] init];

    //  Start the dictionary "churning", repeatedly replacing the
    //  sole value with a new one under the same key.
    [NSThread detachNewThreadSelector: threadSelector
                toTarget: aDictionary
                withObject: nil];

    doGets();
    doSets();

    return 0;
}
```

Listing 2. main2.m (available in the Downloads section) The above source code changes only a couple of lines from the previous example, but no longer crashes.

In this version the `doGets()` and `doSets()` functions add or change code to protect themselves. The `doGets()` function temporarily retains the object it gets from the dictionary, then balances that retention with an autorelease. The "sets" function changes its `release` to an `autorelease`.

Notice that the thread that "churns" the dictionary has the same code as before. It doesn't retrieve an object from the dictionary, and it doesn't use the value it stores in the dictionary.

Why not put the burden on this thread to safeguard the data? Why not have it retain-and-autorelease the object it's about to remove? The answer lies in the fact that each thread has its own autorelease pool. If the "churning" thread sends retain-then-autorelease to the object, that autorelease will take effect when the churning thread frees **its** autorelease pool — that might happen while another thread is still using it.

To sum up:

- Retain-and-release works to temporarily prevent an object from getting deallocated, but you must make sure that `release` gets sent.
- Retain-and-**auto**release works better, because the autorelease pool remembers to send the `release` for you.
- The thread that wants to protect the object must do the autorelease; other threads might wind up releasing the object at the wrong time.

Back to top

Example 3: A category to simplify safety

The above example works, but each piece of code that uses the dictionary has to perform the same steps to use it safely. To make it easy for developers to write safe code, you can encapsulate the functionality for them. One way to do this is to add a category to `NSMutableDictionary`, adding methods that do most of the work for you.

The three methods are:

```
- (id) threadSafeObjectForKey: (id) aKey  usingLock: (NSLock *) aLock;
- (void) threadSafeRemoveObjectForKey: (id) aKey  usingLock: (NSLock *) aLock;
- (void) threadSafeSetObject: (id) anObject
                       forKey: (id) aKey  usingLock: (NSLock *) aLock;
```

For example, if your old code was:

```
        [aDictionaryLock lock];
        anObject = [aDictionary objectForKey: KEY];
        [[anObject retain] autorelease];
        [aDictionaryLock unlock];
```

After adding the category, you can change this to one statement:

```
anObject = [aDictionary threadSafeObjectForKey: KEY
                usingLock: aDictionaryLock];
```

The following source code includes the interface declaration and the implementation for this category, along with the revised application code. (You'll usually put a category's interface in its own .h file and implementation in its own .m file. This example puts everything in a single file for conciseness.)

```
#import <Foundation/Foundation.h>

////////////////////////////////////////////////////////////////
////    NSMutableDictionary CATEGORY FOR THREAD-SAFETY
////////////////////////////////////////////////////////////////

@interface NSMutableDictionary (ThreadSafety)

- (id) threadSafeObjectForKey: (id) aKey
    usingLock: (NSLock *) aLock;

- (void) threadSafeRemoveObjectForKey: (id) aKey
    usingLock: (NSLock *) aLock;

- (void) threadSafeSetObject: (id) anObject
    forKey: (id) aKey
    usingLock: (NSLock *) aLock;

@end


@implementation NSMutableDictionary (ThreadSafety)

- (id) threadSafeObjectForKey: (id) aKey
    usingLock: (NSLock *) aLock;
{
    id  result;

    [aLock lock];
    result = [self objectForKey: aKey];
    [[result retain] autorelease];
    [aLock unlock];

    return result;
}

- (void) threadSafeRemoveObjectForKey: (id) aKey
    usingLock: (NSLock *) aLock;
{
    [aLock lock];
    [self removeObjectForKey: aKey];
    [aLock unlock];
}

- (void) threadSafeSetObject: (id) anObject
    forKey: (id) aKey
    usingLock: (NSLock *) aLock;
{
    [aLock lock];
    [[anObject retain] autorelease];
    [self setObject: anObject  forKey: aKey];
    [aLock unlock];
}
```

```
@end


//////////////////////////////////////////////////////////////
////    TEST PROGRAM
//////////////////////////////////////////////////////////////

static NSMutableDictionary      *aDictionary = nil;
static NSLock                           *aDictionaryLock = nil;

@implementation NSMutableDictionary (Churning)

#define KEY             @"key"

- (void) churnContents;
{
    unsigned long       i;

    for (i = 0; ; i++)
    {
        NSAutoreleasePool       *pool;

        pool = [[NSAutoreleasePool alloc] init];
        [self threadSafeSetObject: [NSString stringWithFormat: @"%d", i]
            forKey: KEY  usingLock: aDictionaryLock];
        [pool release];
    }
}

@end

#define COUNT   10000

static void doGets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
        //      Get the dictionary's value, and then try to message the value.
        [[aDictionary threadSafeObjectForKey: KEY
        usingLock: aDictionaryLock] description];
}

static void doSets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
    {
        NSObject        *anObject;

        anObject = [[NSObject alloc] init];
        [aDictionary threadSafeSetObject: anObject
            forKey: KEY
            usingLock: aDictionaryLock];
        [anObject release];
        [anObject description];
    }
}

int main ()
{
    SEL         threadSelector;

    [[NSAutoreleasePool alloc] init];

    threadSelector = @selector(churnContents);

    aDictionary = [NSMutableDictionary dictionary];
    aDictionaryLock = [[NSLock alloc] init];
```

```
    //  Start the dictionary "churning", repeatedly replacing the
    //  sole value with a new one under the same key.
    [NSThread detachNewThreadSelector: threadSelector
                toTarget: aDictionary
                withObject: nil];

    doGets();
    doSets();

    return 0;
}
```

Listing 3. main3.m (available in the Downloads section)

These methods require you to specify the lock that controls the dictionary. You'd usually choose to use one `NSLock` instance for each dictionary.

A variation on the above approach is to have the category supply simpler methods, such as `threadSafeObjectForKey:`, which supply the lock themselves. This technique has problems because either it uses a single lock for all dictionaries or it has a data structure mapping each dictionary to its respective lock. The data structure will need its own lock, so either way a single lock can become a bottleneck for all threads.

To easily associate each dictionary with a lock, you can subclass the dictionary class. The next example describes in detail how to do this.

Back to top

Example 4: A subclass of NSMutableDictionary

Instead of adding methods like `threadSafeObjectForKey:` in a category, and requiring all developers working on an application to use that method, another technique is to create a subclass of `NSMutableDictionary`, overriding methods like `objectForKey:` and replacing them with thread-safe implementations.

`NSMutableDictionary` is part of the `NSDictionary` class cluster. Subclassing within a class cluster is somewhat complicated, and you shouldn't do it without good reason. For an introduction to both class clusters and ways to subclass them, see the primer on Class Clusters.

The implementation below uses the "composite object" technique described in the class cluster documentation. It defines an object which includes both a real mutable dictionary and a lock. It also implements each of the "primitive" methods in `NSMutableDictionary`.

```
#import <Foundation/Foundation.h>

////////////////////////////////////////////////////////////////
////    NSMutableDictionary SUBCLASS
////////////////////////////////////////////////////////////////

@interface ThreadSafeMutableDictionary : NSMutableDictionary
{
    NSMutableDictionary *realDictionary;
    NSLock                              *lock;
}

@end

@implementation ThreadSafeMutableDictionary : NSMutableDictionary

//  Primitive methods in NSDictionary

- (unsigned) count;
{
    //  I believe we don't need to lock for this.
    return [realDictionary count];
}

- (NSEnumerator *) keyEnumerator;
{
    NSEnumerator        *result;

    //  It's not clear whether we need to lock for this operation,
```

```
    //  but let's be careful.
    [lock lock];
    result = [realDictionary keyEnumerator];
    [lock unlock];

    return result;
}

- (id) objectForKey: (id) aKey;
{
    id  result;

    [lock lock];
    result = [realDictionary objectForKey: aKey];

    //  Before unlocking, make sure this object doesn't get
    //  deallocated until the autorelease pool is released.
    [[result retain] autorelease];
    [lock unlock];

    return result;
}


//  Primitive methods in NSMutableDictionary
- (void) removeObjectForKey: (id) aKey;
{
    //  While this method itself may not run into trouble, respect the
    //  lock so we don't trip up other threads.
    [lock lock];
    [realDictionary removeObjectForKey: aKey];
    [lock unlock];
}

- (void) setObject: (id) anObject forKey: (id) aKey;
{
    //  Putting the object into the dictionary puts it at risk for being
    //  released by another thread, so protect it.
    [[anObject retain] autorelease];

    //  Respect the lock, because setting the object may release
    // its predecessor.
    [lock lock];
    [realDictionary setObject: anObject  forKey: aKey];
    [lock unlock];
}


//      This isn't labeled as primitive, but let's optimize it.
- (id) initWithCapacity: (unsigned) numItems;
{
    self = [self init];
    if (self != nil)
        realDictionary = [[NSMutableDictionary alloc] initWithCapacity: numItems];

    return self;
}

//      Overrides from NSObject
- (id) init;
{
    self = [super init];
    if (self != nil)
        lock = [[NSLock alloc] init];

    return self;
}

- (void) dealloc;
{
    [realDictionary release];
```

```
    [lock release];

    [super dealloc];
}

@end


////////////////////////////////////////////////////////////
////    TEST PROGRAM
////////////////////////////////////////////////////////////

static NSMutableDictionary      *aDictionary = nil;

@implementation NSMutableDictionary (Churning)

#define KEY             @"key"

- (void) churnContents;
{
    unsigned long       i;

    for (i = 0; ; i++)
    {
        NSAutoreleasePool       *pool;

        pool = [[NSAutoreleasePool alloc] init];
        [self setObject: [NSString stringWithFormat: @"%d", i]  forKey: KEY];
        [pool release];
    }
}

@end

#define COUNT   10000

static void doGets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
        //      Get the dictionary's value, and then try to message the value.
        [[aDictionary objectForKey: KEY] description];
}

static void doSets (void)
{
    long        i;

    for (i = 0; i < COUNT; i++)
    {
        NSObject        *anObject;

        anObject = [[NSObject alloc] init];
        [aDictionary setObject: anObject  forKey: KEY];
        [anObject release];
        [anObject description];
    }
}

int main ()
{
    SEL         threadSelector;

    [[NSAutoreleasePool alloc] init];

    threadSelector = @selector(churnContents);

    aDictionary = [ThreadSafeMutableDictionary dictionary];

    //  Start the dictionary "churning", repeatedly replacing the
```

```
    //  sole value with a new one under the same key.
    [NSThread detachNewThreadSelector: threadSelector
                toTarget: aDictionary
                withObject: nil];

    doGets();
    doSets();

    return 0;
}
```

Listing 4. main4.m (available in the Downloads section)

A subclass like this entails several concerns:

- the time to perform locking may hurt performance
- on top of that, any subclass may not be as fast as Apple's "concrete" implementation, which is typically highly optimized
- each object is implemented with two underlying objects, using more memory
- you need to make sure that the dictionary is instantiated from the subclass — notice that in this final version, the dictionary gets created with this code:

```
    aDictionary = [ThreadSafeMutableDictionary dictionary];
```

To sum up:

- Categories provide a simple way to encapsulate new functionality on existing classes.
- Subclassing provides another way to add functionality, but by overriding methods instead of adding them.
- Subclassing within a class cluster requires special attention.

Back to top

Summary

We've looked at four ways to fix thread-safety problems with `NSMutableDictionary` instances (similar approaches would work for `NSMutableArray` and `NSMutableSet`). Each of them had their own advantages and disadvantages. Keep the disadvantages in mind, and use these classes only when operating in a multi-threaded environment. To review, the advantages and disadvantages are:

Example 1 uses a lock to mediate access to the mutable dictionary
Advantage: Clear, simple code
Disadvantage: Like... duh? It crashes!

Example 2 uses `retain` and `autorelease` to protect the object
Advantage: Doesn't crash
Disadvantages: Client code must follow certain rules to be safe
autoreleasing may cost extra time and, temporarily, extra space

Example 3 adds safe methods, encapsulated in a category
Advantages: Doesn't crash; client code is simpler
Disadvantages: Client code must use new methods and supply a lock object
autoreleasing may cost extra time and, temporarily, extra space

Example 4 makes `NSMutableDictionary` methods safe with a subclass
Advantages: Doesn't crash; client code needs no changes
Disadvantages: Dictionary must get instantiated from new class
performance may not match Apple's implementation
autoreleasing may cost extra time and, temporarily, extra space

Back to top

References

Overview of Programming Topic: Multithreading

Thread Safety

Using Foundation from Multiple Threads

Overview of Core Foundation

Back to top

Downloadables

	Acrobat version of this Note (80K)	Download
	Binhexed Sample Code (8K)	Download

Back to top
