Friday Q&A 2012-02-17: Ring Buffers and Mirrored Memory: Part II

by Mike Ash

Last time on Friday Q&A, I started talking about implementing a ring buffer using virtual memory tricks to mirror memory. The first article concentrated on those virtual memory tricks. Today, I'm going to fill out the second half of the puzzle and show how to implement the ring buffer on top of the mirrored memory allocator we developed. If you haven't read the previous article yet, I strongly recommend you so so, otherwise the memory mirroring is likely to be confusing.

Code

Just like last time, the code we're going to discuss is available on GitHub:

https://github.com/mikeash/MAMirroredQueue

Goals

The mirrored memory trick allows exposing pointers to the interior buffer to the outside world, since both data and free space are always contiguous. The goal then is to create an API which makes this easy to use. In default operation, the ring buffer should also grow to accommodate newly written data. For multithreaded use, the ring buffer's size can be locked, at which point the buffer becomes thread safe for one simultaneous reader and writer. For that case, thread safety should be acheived with no locks.

API

The ring buffer is implemented in a class called `MAMirroredQueue`:

```
@interface MAMirroredQueue : NSObject
```

For reading, there are three methods. One method retrieves how much data is available to be read, one method returns a pointer to the data, and one method advances the pointer:

```
- (size_t)availableBytes;
- (void *)readPointer;
- (void)advanceReadPointer: (size_t)howmuch;
```

This way a client can find out how much data it can read, it can access the data, and then when it's finished it can remove that data from the ring buffer by advancing the pointer.
For writing data, the interface is similar. However, instead of a method to query the amount of data, there's a method to simply ensure that the necessary amount of free space is available:

```
- (BOOL)ensureWriteSpace: (size_t)howmuch;
- (void *)writePointer;
- (void)advanceWritePointer: (size_t)howmuch;
```

The `ensureWriteSpace:` method returns success or failure. When allocation is not locked (the default state), it will always succeed. When allocation is locked (to ensure thread safety), it will succeed only if there is enough free space in the buffer, and return `NO` otherwise.
The other two methods in write API are the same as in the read API: one to retrieve the data pointer, and one to advance it once data is written.
With all this talk of locking allocations, we probably want some methods to actually manage that:

```
- (void)lockAllocation;
- (void)unlockAllocation;
```

The queue starts out unlocked. If thread-safe operation is needed, the desired amount of space can be allocated by calling `ensureWriteSpace:`, and afterwards `lockAllocation` ensures that the buffer doesn't get reallocated. If the buffer ever needs to be expanded in the future, `unlockAllocation` followed by another `ensureWriteSpace:` can be used to accomplish that.
Finally, I also wrote a pair of UNIX-like wrappers around the above functionality:

```
- (size_t)read: (void *)buf count: (size_t)howmuch;
- (size_t)write: (const void *)buf count:
(size_t)howmuch;
```

These aren't necessary, and are actually inefficient because the API requires copying data into and out of the ring buffer, but an API that shares the semantics of the POSIX `read` and `write` calls can be nice to have.

Instance Variables

The buffer itself is described by three instance variables:

```
    char *_buf;
    size_t _bufSize;
    BOOL _allocationLocked;
```
All of which are, I hope, self explanatory. Note that `_buf` is a `char *` to allow for easy pointer arithmetic, which will come in handy later. Although `gcc` and `clang` allow it, technically pointer arithmetic is not allowed on `void *` pointers, so `char *` is a convenient choice for byte-addressed entities.
In addition to the buffer, we also need a read pointer and a write pointer:
```
    char *_readPointer;
    char *_writePointer;
```
Utility Functions
Page size is important for this code, and it needs to be able to round numbers up and down to a multiple of the page size. I wrote two simple wrappers around mach macros which manage this:
```
    static size_t RoundUpToPageSize(size_t n)
    {
        return round_page(n);
    }

    static void *RoundDownToPageSize(void *ptr)
    {
        return (void *)trunc_page((intptr_t)ptr);
    }
```
The first one simply rounds a byte size up to the nearest multiple of page size, and the second one rounds a pointer down to the nearest page boundary.
Code
First up is the `dealloc` method. I'm assuming `ARC`, so no need to call `super`. All it does is free the buffer if it's been allocated:
```
    - (void)dealloc
    {
        if(_buf)
            free_mirrored(_buf, _bufSize, MIRROR_COUNT);
    }
```
`MIRROR_COUNT` is simply a `#define` which describes how many mirrored copies to allocate. Interestingly, it's set to `3`, not `2` as you might expect, which is why my mirrored allocator supports an

arbitrary number of mirrorings instead of just hardcoding two of them. More on the reasoning for this later.
There is no initializer method, as simply having all of the instance variables set to `0` or `NULL` is sufficient. The ring buffer starts out empty, and all zeroes describes that just fine.
Next up, we have the `availableBytes` method. The first thing it does is subtract the read pointer from the write pointer:

```
- (size_t)availableBytes
{
    ptrdiff_t amount = _writePointer - _readPointer;
```

Normally, this would just be the number of data bytes in the buffer. However, in the event that another thread is modifying this buffer while we're computing this value, the pointers could be moving around. If they're just moving around by a read or write amount, then that's no problem. We may end up computing the old value or the new value for the number of available bytes, but either one works fine.
However, the pointers can also be moved around by the size of the buffer. When the read pointer goes into the second mirrored region, it gets reset back into the first one, and the write pointer follows. Because of this, the computed size here may be less than zero (if we see the update to the write pointer but not the read pointer), or may be greater than the buffer size (if we see the read pointer update but not the write pointer). Since we know that the number of available bytes *must* be between zero and the buffer size, this is easy to correct: just check for these cases, and adjust the amount accordingly:

```
    if(amount < 0)
        amount += _bufSize;
    else if((size_t)amount > _bufSize)
        amount -= _bufSize;

    return amount;
}
```

Next up, we have `readPointer`. This simply returns the ivar:

```
- (void *)readPointer
{
    return _readPointer;
}
```

Next, `advanceReadPointer:`. This simply adds the amount to the read pointer:

```
- (void)advanceReadPointer: (size_t)howmuch
{
    _readPointer += howmuch;
```

However, it's not done here. In the event that this advanced the read pointer past the end of the first mirrored region, both it and the write pointer need to be pulled back. For the read pointer, this is simply a matter of subtracting `_bufSize`. Since the write pointer can be modified by both the reader thread (with this method) and the writer thread (in `advanceWritePointer:`) simultaneously, it needs to be updated using an atomic operation. I use the built-in `__sync_sub_and_fetch` function to do this:

```
    if((size_t)(_readPointer - _buf) >= _bufSize)
    {
        _readPointer -= _bufSize;
        __sync_sub_and_fetch(&_writePointer, _bufSize);
    }
}
```

Next comes `ensureWriteSpace:`. The first part of this is trivial: find out how much free space is available, by subtracting `[self availableBytes]` from the total buffer size, and if the requested amount is less, everything is all set:

```
- (BOOL)ensureWriteSpace: (size_t)howmuch
{
    size_t contentLength = [self availableBytes];
    if(howmuch <= _bufSize - contentLength)
        return YES;
```

Otherwise, we know that the free space is *not* sufficient to meet the request. If allocation is locked, that's it, game over, return `NO`:

```
    else if(_allocationLocked)
        return NO;
```

If allocation is not locked, then it's time to reallocate the buffer. The first thing to do is figure out how much memory to allocate, then allocate a new buffer of that size. Recall that, because the mirrored allocator uses virtual memory tricks, it must allocate a multiple of the page size. We need at least `contentLength + howmuch` memory, so the new buffer size is found by rounding that number up to the nearest page size:

```
        size_t newBufferLength =
RoundUpToPageSize(contentLength + howmuch);
```
Next, allocate the new buffer:
```
        char *newBuf = allocate_mirrored(newBufferLength,
MIRROR_COUNT);
```
Now that we have the new buffer, the code branches a bit. If there's already an existing buffer, then we're reallocating memory, and have to copy the data out of the old buffer and into the new:
```
        if(_bufSize > 0)
        {
```
Once again, we're going to play virtual memory games. Mach provides a `vm_copy` function which copies page-aligned memory without actually copying it. Instead, the pages are remapped and set up to copy on write. For this case, where we're going to immediately deallocate the old memory, this means that no data is ever actually copied, and the system just plays some virtual memory tricks to make it look like it was.
We want to copy starting from the read pointer, but because everything has to be page-aligned, the copy has to start at the beginning of the page containing the read pointer:
```
            char *copyStart =
RoundDownToPageSize(_readPointer);
```
Likewise, the length needs to be a multiple of the page size. Starting from `copyStart`, we need to copy `_writePointer - copyStart` bytes, but this needs to be rounded up fit the page size:
```
            size_t copyLength =
RoundUpToPageSize(_writePointer - copyStart);
```
Now that this is set, we can "copy" this data into the new buffer:
```
            vm_copy(mach_task_self(),
(vm_address_t)copyStart, copyLength, (vm_address_t)newBuf);
```
Now that the data is copied, we need to compute the location of the new read pointer. We copied additional bytes by rounding `_readPointer` down to `copyStart`. The new read pointer is equal to `newBuf` plus that number of additional bytes:
```
            char *newReadPointer = newBuf + (_readPointer -
copyStart);
```
This spot was particularly troublesome for me when I was developing the code, so I tossed in an assert to make sure that it failed early and loudly:

```
            if(*newReadPointer != *_readPointer)
                abort();
```
Now we can free the old buffer and reassign the read pointer:
```
            free_mirrored(_buf, _bufSize, MIRROR_COUNT);
            _readPointer = newReadPointer;
```
The write pointer is set to equal the read pointer plus the previously computed content length:
```
            _writePointer = _readPointer + contentLength;
        }
```
For the case where no previous buffer exists, the code is simple: just set the read and write pointer to the beginning of the new buffer:
```
        else
        {
            _readPointer = newBuf;
            _writePointer = newBuf;
        }
```
The new buffer is allocated, data is copied if necessary, and now all that remains is to set the `_buf` and `_bufSize` ivars and then return `YES` to the caller:
```
        _buf = newBuf;
        _bufSize = newBufferLength;

        return YES;
    }
```
Next up, the `writePointer` method, which is once again just a simple accessor:
```
    - (void *)writePointer
    {
        return _writePointer;
    }
```
The `advanceWritePointer:` method is also simple, performing an atomic add of `_writePointer`:
```
    - (void)advanceWritePointer: (size_t)howmuch
    {
        __sync_add_and_fetch(&_writePointer, howmuch);
    }
```
Note that, unlike `advanceReadPointer:`, this method doesn't need any checks to wrap pointers back to the first mirrored data section. The `advanceReadPointer:` method handles wrapping both read and write pointers. Since no wrapping occurs here, the

write pointer can often sit in the second mirrored data section for long periods of time, but that's perfectly fine.
By having the write pointer always be ahead of the read pointer, this code avoids an annoying ambiguity. A ring buffer which uses read and write pointers usually suffers from a problem when the read and write pointers are equal. There's no simple way to distinguish between the buffer being empty (where both pointers are equal because there is no data between them) and the buffer being full (where the pointers are equal becaus there's no free space between them).
Ring buffer implementations typically avoid this either by prohibiting the buffer from becoming completely full, and instead defining "full" as being one unit less than true full capacity, or by using a pointer plus a count rather than using two pointers.
Neither alternative is attractive in this case. Having the buffer be artifically one byte small is particularly painful when playing crazy virtual memory games, since it's likely that code using this ring buffer will want to deal in whole pages as well, and by losing a single byte out of the buffer, it essentially loses an entire 4kB page.
Using a pointer and a count makes achieving lockless thread safety much more difficult if not completely impossible, since the write pointer becomes a derived value computed from two other values, both of which get modified by the read thread, and which can be momentarily inconsistent with each other as seen from another thread when both are updated simultaneously.
The virtual memory games played with mirrored allocation give a third, better way: simply use two pointers, and express "full" by having the write pointer equal the read pointer plus the buffer size. It's natural, it works great due to the mirrored allocation, and it's easy to deal with.
Next up are the allocation locking methods. These simply manipulate the `_allocationLocked` ivar. Nothing else needs to be done in these methods, since they just modify the behavior of `ensureWriteSpace:`. Here's the code:

```
- (void)lockAllocation
{
    _allocationLocked = YES;
```

```
    }

    - (void)unlockAllocation
    {
        _allocationLocked = NO;
    }
```

Next, we have the UNIX-like compatibility wrappers. These help illustrate how the more primitive, direct API is used. The `read:count:` method figures out how much to read using `availableBytes`, copies data out from `readPointer`, then calls `advanceReadPointer:` to mark the data as read:

```
    - (size_t)read: (void *)buf count: (size_t)howmuch
    {
        size_t toRead = MIN(howmuch, [self availableBytes]);
        memcpy(buf, [self readPointer], toRead);
        [self advanceReadPointer: toRead];
        return toRead;
    }
```

The story for `write:count:` is a little more complex, as it behaves differently depending on whether allocation is locked. If allocation is locked, then it only writes as much data as will fit in the buffer's remaining space. Otherwise, it uses `ensureWriteSpace:` to grow the buffer to the appropriate size, if needed:

```
    - (size_t)write: (const void *)buf count: (size_t)howmuch
    {
        if(_allocationLocked)
            howmuch = MIN(howmuch, _bufSize - [self
availableBytes]);
        else
            [self ensureWriteSpace: howmuch];
```

The rest is straightforward. It copies the data into `writePointer`, advances the write pointer, and returns the amount of data written:

```
        memcpy([self writePointer], buf, howmuch);
        [self advanceWritePointer: howmuch];
        return howmuch;
    }
```

And that completes the implementation of the mirrored queue.

Thread Safety

One of the goals of this implementation is to be thread safe, for the case where there is one reader thread and one writer thread. The

above code achieves this, but it's not entirely clear why it's safe. There are no locks, and the read pointer doesn't even use atomic operations to update.

First, note that the code is only thread safe for the case when allocation is locked. That means that all of the tricky reallocation code in `ensureWriteSpace:` doesn't come into play. This is good, because it would be incredibly difficult to make that code thread safe without locking. Given that, we can consider `_buf` and `_bufSize` to be constant. The only variables which could be potentially modified by one thread while simultaneouly being read by another thread are `_readPointer` and `_writePointer`.

It's easiest to consider this as two separate cases. First, the reader thread needs to be correct even when the writer thread is modifying these values. Second, the writer thread needs to be correct even when the reader thread is modifying these values. If both hold, then the entire hting is correct.

Let's look at the first case, of making sure the reader thread is correct in the face of modifications from the writer thread. The writer thread only ever modifies `_writePointer`. The only place where the reader thread depends on the value of `_writePointer` is in `availableBytes`:

```
    ptrdiff_t amount = _writePointer - _readPointer;
```

This unsynchronized access is perfectly safe. There's a race condition in that it's not certain whether the read thread will see the old or new value of `_writePointer`. However, it doesn't matter. `_writePointer` can only increase, and the number of available bytes can likewise only increase. If it sees the old value, it still computes a number of available bytes that's *correct*, just slightly out of date. If it sees the new value, then so much the better. Therefore, the reader is safe.

Let's look at the writer's safety in the face of changes from the reader now. The reader can alter both pointers, so the analysis is a little more complex. The writer code also calls `availableBytes`, a method nominally on the reader side, so that method has to be safe for both.

The only way that the reader thread can alter `_writePointer` is with this line:

```
    __sync_sub_and_fetch(&_writePointer, _bufSize);
```

Because of the mirrored structure of the underlying buffer, both the old and new values of `_writePointer` are correct here. `availableBytes` can deal with either value, and will return the right answer in either case. Likewise, when `writePointer` returns the value, it doesn't matter whether it's the old or new value, as they are both equivalent in terms of what happens when data is written to them. Finally, `advanceWritePointer:` is safe, since it also uses a `__sync` builtin to modify `_writePointer`, ensuring that both updates will be applied in *some* order, and the particular order is not important.

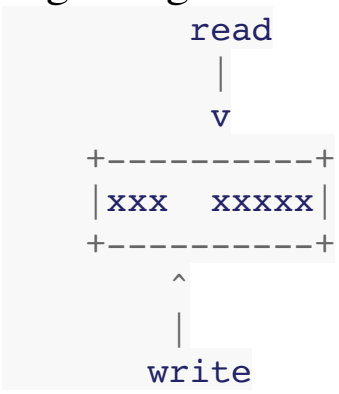
The only place where the writer thread uses `_readPointer` is in `availableBytes`. Just like the corresponding case when the writer thread modifies its pointer and the reader calls `availableBytes`, it's safe for the reader thread to modify `_readPointer` while the writer is calculating `availableBytes`. Advancing the read pointer decreases the number of available bytes, which *increases* the amount of write space. If the writer thread sees the old value for `_readPointer` here, it computes the older, smaller amount of write space, which is still safe, just slightly stale. The writer thread is safe in the face of changes from the reader thread, and vice versa. Therefore, this code really is thread safe.

Triplicate

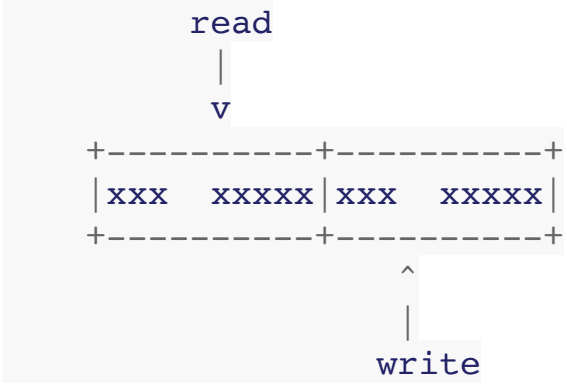
I promised that I would explain why `MAMirroredQueue` allocates *three* mirrored copies of its buffer, and now is the time.

Normally, two copies is enough for the contiguous ring buffer. However, remember that this implementation is a little odd, even for a mirrored ring buffer, in using separate read and write pointers and allowing the write pointer to sit in the second mirrored area. This enables lockless thread safety and use of the full buffer without the strange ambiguity when the two pointers are equal. However, it also requires the existence of a third mirrored area.

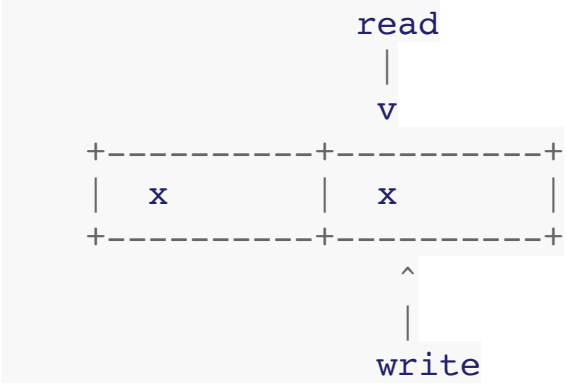
The need is extremely rare, but it can happen. To start with, we need a buffer where the data portion has wrapped around to the beginning. In a normal ring buffer, that would look like this:



In the mirrored ring buffer, it instead looks like this:

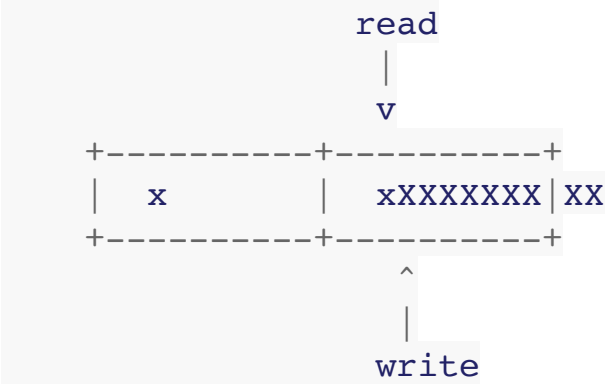


So far so good, still. The write pointer is valid, and writing data into the empty area is correct. Now, let's say there are separate reader and writer threads manipulating this buffer simultaneously. The reader thread moves the read pointer up:



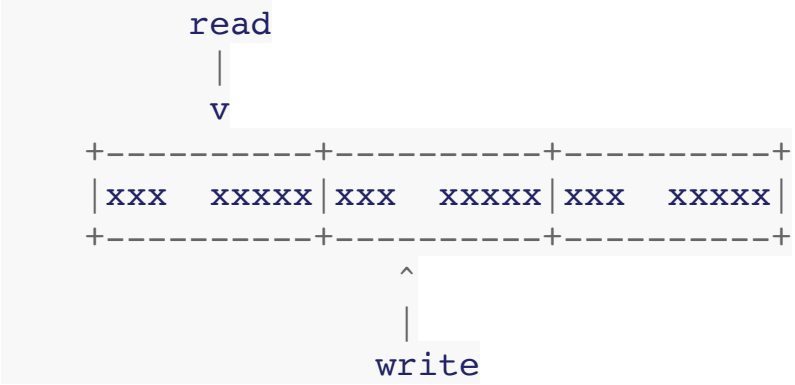
The next step for the reader thread is to move both read and write pointers down into the first mirrored area. *But!* Before that can happen, let's say that the writer suddenly goes to write a bunch of data. (Remember, in the world of preemptive threading, the two threads can run in all sorts of weird orders.) Before the read thread

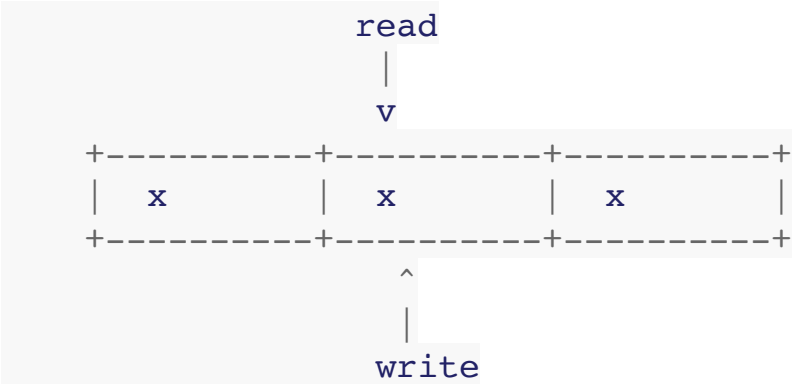
can move any pointers around, the write thread computes the available space, writes data into the buffer, and crashes:

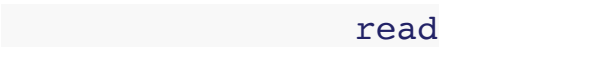


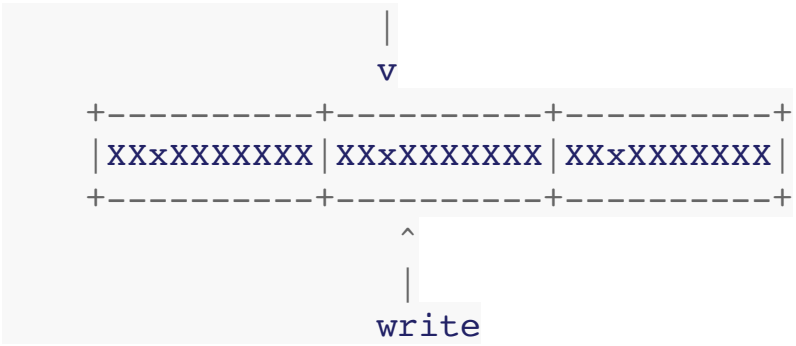
The writer wrote off the end of the second mirrored segment! It saw a large amount of free space available, which is correct, but the trouble is that the end of this free space can only be accessed from a write pointer residing in the first mirrored segment. Since we allow the write pointer to hang out in the second mirrored segment, there's the possibility for trouble. This is a brief race window requiring very specific circumstances to trigger, but it can happen, and it needs to be protected against.

Fortunately, this problem is easy to solve by simply allocating a third mirrored segment. In that case, the sequence looks like this:









With the extra mirrored region at the end, the surplus data is written to a safe spot and everything works correctly. At this point, the read thread will jump in, shift the pointers down, and life continues as usual.

Conclusion

That wraps up our exploration of a mirrored-memory ring buffer. I hope the journey was enlightening. Oh, and the source code is all available under a standard BSD license in case you have a use for it in your own apps.

Come back next time for another exciting episode. Friday Q&A is, as always, driven by reader input, so send in your ideas for topics in the meantime.

Comments:

Peter Bierman at 2012-05-12 01:20:32:

Is the code thread-safe for multiple reader and/or writer threads? I have a guess as to how to add that, but guessing and thread-safety don't mix well.

mikeash at 2012-05-12 02:46:43:

It's not, no. The API as-is is inherently unsafe for that scenario, since reading/writing are separate from advancing the pointer. I'm not sure offhand how you'd make it safe for multiple readers or writers, but it would probably have to start with an atomic get-and-advance-pointer operation, and some more bookkeeping to distinguish between areas that are data, free, or pending data.

Peter N Lewis at 2012-06-24 06:54:06:

Nice article as always.

s/hting/thing/