Friday Q&A 2012-02-03: Ring Buffers and Mirrored Memory: Part I

by Mike Ash

Playing with virtual memory is always fun, and one place where it can be put to good use is in building a ring buffer. A ring buffer is a way to implement a FIFO queue of data, and using virtual memory tricks to mirror multiple copies of the same data can make the implementation simpler and better by virtually concatenating noncontiguous data. Readers Jose Vazquez and Dylan Lukes suggested that I explore the building of such a construct. Today I will talk about how to implement the virtual memory tricks, and then in part II I will show how to implement the ring buffer on top of them.

Ring Buffer

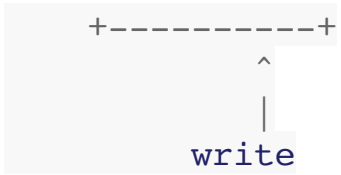
A ring buffer is a way to implement a fast FIFO (first in, first out) queue. It works by allocating a chunk of memory and tracking read and write pointers. Those pointers advance as data is read and written. The part that makes it a *ring* buffer is that, when one of those pointers goes off the end, it wraps back to the beginning.

To illustrate, the ring buffer starts out empty:

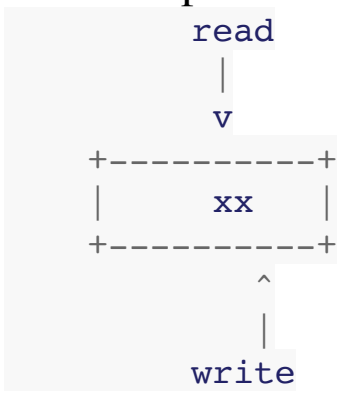
```
    read
     |
     v
    +----------+
    |          |
    +----------+
     ^
     |
    write
```

When data is written, the data fills out the buffer and the write pointer is advanced:

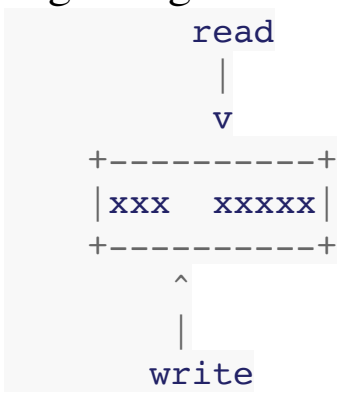
```
    read
     |
     v
    +----------+
    |xxxxxxx   |
```



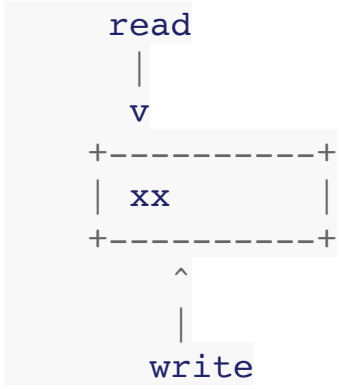
The read pointer follows when data is read:



When the write pointer goes off the end, it wraps back to the beginning:



At this point, the data in the ring buffer is no longer contiguous, as it wraps off the end and back to the beginning. The ring buffer code has to split the data apart when writing and then stitch it back together when reading it back out again. After reading some more, the read pointer also wraps around and the buffer looks like this:



And it can continue like this forever, wrapping around each time it hits the end.

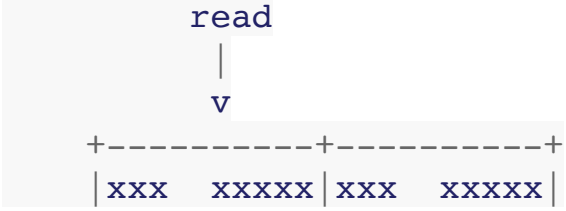
This structure is great for anything where data is continuously written and then read shortly afterwards. It's relatively easy to make thread safe, so it can be especially useful for communication of realtime data between threads. Ring buffers are commonly used for communicating audio into playback threads and out of recording threads, for example.

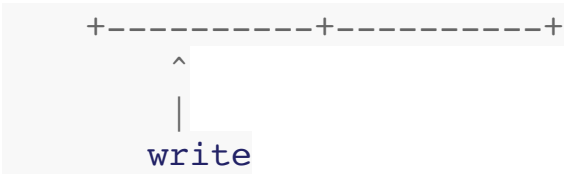
Mirroring

This technique works well and is used a lot. However, the wraparound behavior makes things a bit difficult. It complicates the reading and writing code, as it has to check for wraparound and then split the read/write in two. Worse, however, is the fact that the wraparound forces a memory copy for getting data into or out of the ring buffer. Since the data isn't necessarily contiguous, it's not possible to expose pointers to the data to the outside world. For example, if you're reading into the ring buffer from a file, it would be great to be able to pass the ring buffer's write pointer directly to the `read` function, but this won't easily work if the free space wraps around. Instead, you either have to complicate things with multiple reads, or read into a separate buffer and then copy the memory over. Imagine if it were possible to mirror a chunk of memory, such that it appeared in two places at once. Any change to one place would be immediately reflected in the other place. For example, you'd start out empty:

Then upon writing to one copy, the other copy immediately sees the data as well:

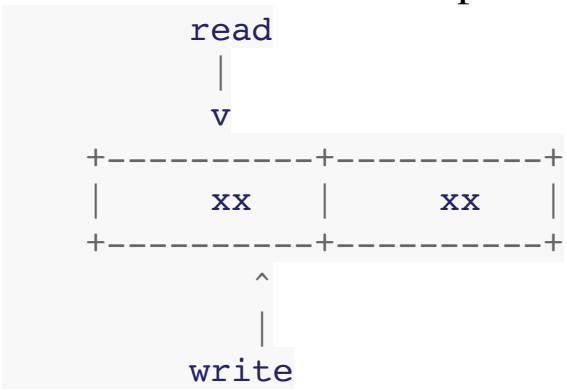
Let's do this to the ring buffer's memory. Furthermore, let's arrange it so the two mirrored copies are directly side by side:





Check it out! The data is now contiguous despite wrapping around in the ring buffer, because the second copy directly adjoins the first. Even though the data wraps around, we can still pass a pointer to it to anything that expects contiguous data. Even though the data wraps around, it *is* contiguous, thanks to mirroring.

The same thing applies for the empty space, for the case when the buffer's data doesn't wrap around:



Again, the free space is contiguous, simplifying things greatly. This pointer can be passed directly to a `read` call or anything else that wants to write into a single block of memory.

Code

It's time to dive into code. As usual, the code for today's adventure is available on GitHub:

https://github.com/mikeash/MAMirroredQueue

Mirroring With Mach

Being able to mirror memory like this would be really handy. Well, good news! With the mach virtual memory APIs, you can!

When dealing with virtual memory, everything relates to memory pages. A page is a chunk of memory of a convenient size to be handled by the virtual memory system, which is typically 4096 bytes. A page is the smallest unit that virtual memory will deal with. If you map or unmap memory, you can only deal with an integral number of pages.

Although the page size will be 4096 bytes on any system you're likely to encounter, it's not a good practice to hardcode this. The system's page size can be obtained from the `vm_page_size` global

variable. Here's a simple wrapper that hides the global behind a function call:

```
size_t get_page_size(void)
{
    return vm_page_size;
}
```

Now, let's talk about how to implement the actual mirrored memory allocator. The mach virtual memory functions allow allocating a chunk of memory, remapping a chunk of memory to a specified location, and deallocating a chunk of memory. Crucially, unlike `malloc` and `free`, it's possible to deallocate only a piece of an allocation. In other words, you can allocate two pages, then only deallocate one, and the other page remains valid.

This is what allows us to obtain a contiguous chunk of mirrored memory. If we just allocated a chunk of memory and then tried to remap it to the space afterwards, this is likely to fail, as that space may well be occupied. There's no way to directly ask the system to allocate memory in an area that's followed by a bunch of free space. However, this can be done indirectly. Here's the approach:

1. Allocate twice as much memory as required.
2. Deallocate the second half of the memory that was just allocated.
3. Remap the first half into the now free second half.

There's a race condition here, where another thread could end up allocating memory in the newly freed second half between steps 2 and 3. However, this isn't a disaster. If it happens, step 3 will fail, and we can just deallocate the memory and try the whole operation again. Since the race window is small, it's unlikely to require many tries for this procedure to succeed.

Let's look at the code to do it. Rather than only implement mirrored pairs, I made this function able to allocate an arbitrary number of contiguous mirrored memory chunks. This function takes a quantity of memory to allocate and a number of mirrored copies to make. The quantity of memory to allocate has to be an integral number of pages, and to keep things from blowing up weirdly later on, I check for that and return an error if it doesn't match:

```
void *allocate_mirrored(size_t howmuch, unsigned howmany)
```

```
    {
        // make sure it's positive and an exact multiple of
page size
        if(howmuch <= 0 || howmuch != howmuch /
get_page_size() * get_page_size())
        {
            errno = EINVAL;
            return NULL;
        }
```

Now that the size is known to be good, we'll start the loop. The `mem` variable will hold the allocated pointer. We start it out as `NULL` and then keep trying until things work.

```
        char *mem = NULL;
        while(mem == NULL)
        {
```

We're going to be doing a lot of error checking on the results of these mach calls. After the initial chunk of memory is allocated, handling an error means deallocating the memory that's still allocated before returning the error. Before we get to the actual code, we need a macro to centralize this error handling.

The `CHECK_ERR` macro checks an expression against `KERN_SUCCESS` and handles cleanup. It takes a second parameter, which is the number of bytes to deallocate. In the event that the initial allocation fails, no memory needs to be deallocated. Later failures need to either deallocate the entire chunk of memory, including all of the mirrored pages, or just some of them, depending on exactly what is allocated at that time. After deallocating memory, if needed, the macro then sets `errno` and returns `NULL`:

```
            #define CHECK_ERR(expr, todealloc) do { \
                kern_return_t __check_err = (expr); \
                if(__check_err != KERN_SUCCESS) \
                { \
                    if(todealloc > 0) \
                        vm_deallocate(mach_task_self(),
(vm_address_t)mem, (todealloc)); \
                    errno = ENOMEM; \
                    return NULL; \
                } \
            } while(0)
```

With this in place we can finally make the first mach call of this function, to make the initial memory allocation:

```
            CHECK_ERR(vm_allocate(mach_task_self(),
(vm_address_t *)&mem, howmuch * howmany, VM_FLAGS_ANYWHERE),
0);
```

No memory needs to be freed if this allocation fails, so the second parameter to the macro is `0`. Next, we'll deallocate all of this newly allocated space beyond the first chunk. First, we calculate the location of the remainder by adding `howmuch` to the newly allocated pointer, then using `vm_deallocate` to deallocate `howmany - 1` copies:

```
            char *target = mem + howmuch;
            CHECK_ERR(vm_deallocate(mach_task_self(),
(vm_address_t)target, howmuch * (howmany - 1)), howmuch *
howmany);
```

If this fails, then we should deallocate the entire allocated region. Realistically, this won't fail, and if it does then the attempt to deallocate the entire region isn't too likely to work either, but it's nice to be thorough.

Next up, we remap the initial chunk into the remaining space. We need to do this in a loop, once for each additional copy:

```
            for(unsigned i = 1; i < howmany && mem != NULL;
i++)
            {
```

Then remap the initial chunk into the current target section:

```
                vm_prot_t curProtection, maxProtection;
                kern_return_t err =
vm_remap(mach_task_self(),
                                                  (vm_address_t
*)&target,
                                                  howmuch,
                                                  0, // mask
                                                  0, // anywhere

mach_task_self(),

(vm_address_t)mem,
                                                  0, // copy
                                                  &curProtection,
                                                  &maxProtection,

VM_INHERIT_COPY);
```

This is a pretty complicated call, so let's look at what all of the parameters mean. The first parameter, `mach_task_self()`, tells it that we want to map memory into our own process and not some other process. The mach calls actually allow manipulating memory mappings in other processes, if you have the necessary privileges. We're not doing anything funny like that here, so we just pass `mach_task_self()`.

The second parameter is the target of the memory remap. If you don't care about the location of the remapped memory, `vm_remap` can choose an available address for you, and it will return that address in this parameter, thus it's passed by reference. In this case, however, we need it to go into a particular spot, so we don't use that option. We still have to pass a pointer, though, since that's part of the API.

The third parameter is how much memory to remap, which is just the `howmuch` parameter to `allocate_mirrored`.

The fourth parameter is a bitmask to specify alignment restrictions for the starting address, when we have `vm_remap` choose it for us. Since we aren't doing that, this parameter doesn't matter and we just pass `0`.

The fifth parameter is a flag saying whether `vm_remap` can choose its own address for the remap, or whether it must remap into the location provided. By passing `0`, we say that it must use the location provided.

The sixth parameter is the source task (process) for the remap operation. Since we're working entirely within our own process, we once again pass `mach_task_self()`. The seventh parameter is the source address, which is just the beginning of the allocated space, stored in `mem`.

The eight parameter is a flag determining whether the memory is to be copied or remapped read-write. The whole purpose of this code is to remap read-write, so we pass `0` to indicate that we don't want a copy.

The next two parameters are out parameters telling us about the read/write/execute protections on the memory region. We don't care

about these, but the documentation doesn't say that it's legal to pass `NULL`, so we pass in pointers to dummy variables.
The last parameter indicates how child processes inherit this region. `VM_INHERIT_COPY` means that child processes get a copy, which is generally the expected behavior. It's unlikely that a child process would ever care about this memory to begin with, so this parameter is not all that important.
That's the monster remap call. Assuming success, the initial chunk of memory is now remapped into the target area. We can then increment the `target` pointer by `howmuch` to position it for the next chunk to be remapped, in the event that we need to do more than one:

```
                target += howmuch;
```

Now that the remap call is done, we need to do error checking. The error checking for this call is a little more complex, because an error due to the space being occupied by something else is *not* fatal. Recall that this whole procedure is betting against a race conditiong occurring, where another thread allocates memory in the mirrored space before we get a chance to remap it. If that happens, it's not a failure, we simply have to try again.
For that case, the error will be `KERN_NO_SPACE`. In that event, we deallocate the original chunk plus whatever has been remapped so far, then set `mem` to `NULL` to have the loop try again:

```
                if(err == KERN_NO_SPACE)
                {
                    CHECK_ERR(vm_deallocate(mach_task_self(),
(vm_address_t)mem, howmuch * i), 0);
                    mem = NULL;
                }
```

For all other cases, the `CHECK_ERR` macro suffices:

```
                else
                {
                    CHECK_ERR(err, howmuch * i);
                }
```

If we get to the end of the loop with `mem` set to something, then success! Nothing left to do but return the new pointer to the caller:

```
            }
        }
        return mem;
```

```
    }
```

Of course, we need a corresponding function to deallocate the mirrored memory. All we need to do is call `vm_deallocate` on the pointer with the correct size. How can we know the correct size? With functions like `malloc` and `free`, some metadata is stashed away to describe how large the memory region is. For this function, we'll just cheat and make the caller pass the size and count in as parameters, just like with the allocation function. This way it's the caller's problem, and they can solve it however is appropriate for them:

```
    void free_mirrored(void *ptr, size_t howmuch, unsigned
howmany)
    {
        vm_deallocate(mach_task_self(), (vm_address_t)ptr,
howmuch * howmany);
    }
```

And that's it! We can now allocate mirrored memory as desired, with an arbitrary number of contiguous mirrorings. Here's a little test code to make sure it works as intended, by writing to part of the mirrored memory, reading from another part, and making sure they always have the same contents:

```
    static void test_size(unsigned howmany, size_t howmuch)
    {
        char *buf = allocate_mirrored(howmuch, howmany);

        unsigned short seed[3] = { 0 };
        for(unsigned j = 0; j < howmany; j++)
        {
            for(size_t i = 0; i < howmuch; i++)
                buf[i] = nrand48(seed);
            if(memcmp(buf, buf + howmuch * j, howmuch) != 0)
                fprintf(stderr, "FAIL: writing to first half
didn't update second half with size %lu\n", (long)howmuch);

            for(size_t i = 0; i < howmuch; i++)
                buf[howmuch * j + i] = nrand48(seed);
            if(memcmp(buf, buf + howmuch * j, howmuch) != 0)
                fprintf(stderr, "FAIL: writing to second half
didn't update first half with size %lu\n", (long)howmuch);
        }

        free_mirrored(buf, howmuch, howmany);
```

```
    }
```

Just call that a few times with different sizes and counts and make sure everything works as expected:

```
    void test_allocate_mirrored(void)
    {
        for(unsigned i = 2; i < 10; i++)
        {
            test_size(i, get_page_size());
            test_size(i, get_page_size() * 2);
            test_size(i, get_page_size() * 10);
            test_size(i, get_page_size() * 100);
        }
    }
```

And indeed, everything comes out as it's supposed to.

Conclusion

With the mirrored allocator up and running, we're ready to build the ring buffer on top of it. Unfortunately, we're out of time, so come back for part II where we'll examine in detail how that side of things works.

Comments:

Jean-Daniel at 2012-02-03 16:57:34:

While using mach API is fun, it's also good to know that this trick can also be written in a more portable way using only POSIX functions (for instance mmap), and moreover it does not suffer from the race condition you have using the mach API.

An implementation using mmap can be found here:

http://en.wikipedia.org/wiki/Circular_buffer#Optimized_POSIX_Implementation

mikeash at 2012-02-03 17:05:07:

That's interesting, I'll have to see how mmap is implemented that lets it atomically replace an existing mapping, since as far as I know it's implemented on top of the mach APIs. I dislike how the POSIX version

requires hitting the filesystem, but obviously that's a better choice if you can't assume the existence of mach.

John McLaughlin at 2012-02-03 20:06:46:

Thanks Mike,

As usual a very nicely done article on an interesting problem.

-John

Jean-Daniel at 2012-02-03 20:43:08:

@Mike: You're right about mmap implemented above mach API. I dislike too having to use the filesystem for such task, and before your comment, I didn't realized mmap was based on mach.

But knowing that, I just managed to write a working mach based version using the following functions:
- vm_allocate(capacity*2, anywhere) to create a large block.
- vm_allocate(capacity, VM_FLAGS_FIXED | VM_FLAGS_OVERWRITE) to create a region in the first half of the block.
- make_memory_entry() to get an port representing this region, and vm_map(VM_FLAGS_FIXED | VM_FLAGS_OVERWRITE) to remap it in the second half of the block.

Rich Pollock at 2012-02-03 23:28:45:

$article ~= s/locatino/location/

Otherwise, it sounds like passing 0 as the fifth argument of vm_remap forces it to use a fancy (if fictitious) coffee rather than choosing its own address.

mikeash at 2012-02-04 02:15:01:

Jean-Daniel: Nifty. It seems unfortunate that vm_remap doesn't have access to those flags. I'm not sure whether your version or mine is better or less complicated. Good to know other ways of doing it, anyway.

Rich Pollock: Having a coffee API in mach would simplify some things, certainly. Fixed the typo, thanks.

Magnus Reftel at 2012-02-06 06:42:21:

Instead, you either have to complicate things with multiple reads, or read into a separate buffer and then copy the memory over.

Well, you could of course also set up a scatter/gather operation via readv. Still, it's nowhere near as cool as the mirroring trick.

Some Guy at 2014-10-06 14:38:48:

actually vm_remap has acces to VM_FLAGS_OVERWRITE and will do what you want, I just tested it does the right thing (in the 'flags' arguments) and is non racy.

a3f at 2017-03-01 15:56:29:

As Some Guy said, the code doesn't have to be racy. Here's an example of using VM_FLAGS_OVERWRITE:

https://github.com/a3f/libvas/blob/master/mach/ringbuf.c