Using black magic to make a fast circular buffer.

15 Aug 2017 • Travis Mick

Yesterday, I took a glance at the Wikipedia page for the circular buffer and was intrigued by an alleged optimization technique that I was not familiar with:

> A circular-buffer implementation may be optimized by mapping the underlying buffer to two contiguous regions of virtual memory. (Naturally, the underlying buffer's length must then equal some multiple of the system's page size.) Reading from and writing to the circular buffer may then be carried out with greater efficiency by means of direct memory access; those accesses which fall beyond the end of the first virtual-memory region will automatically wrap around to the beginning of the underlying buffer. When the read offset is advanced into the second virtual-memory region, both offsets—read and write—are decremented by the length of the underlying buffer

When implementing a circular buffer, we need to handle the case where a message spans the "discontinuity" in the queue and wraps around. The naive circular buffer's write routine might employ a byte-by-byte write and look something like this:

```
void put(queue_t *q, uint8_t *data, size_t size) {
    for(size_t i = 0; i < size; i++){
        q->buffer[(q->tail + i) % q->buffer_size] = data[i];
    }
    q->tail = (q->tail + size) % q->buffer_size;
}
```

The fact that a modulo operation is necessary to index into the array makes this function hard (if not impossible) to vectorize, and thus unnecessarily slow. Though there are other optimizations we can make, the technique offered in the above Wikipedia surpasses hardware-agnostic approaches by virtue of the fact that the memory management unit can handle most of the wrap-around logic on our behalf. I was so excited by this idea that I did no further research whatsoever, and implemented it based only on the brief description above.

The next two sections overview the circular buffer's design and the behavior of virtual memory, respectively. If you don't need the refresher, feel free to skip ahead.

The Circular Buffer

The circular buffer is a convenient approach to the storage of streaming data which is produced in real-time and consumed shortly thereafter. It "wraps around" so that new data may continually reuse the space previously occupied by data which has since been consumed.

A circular buffer is typically implemented by storing two pointers (or indices): a *read pointer* (I'll call it `head`) and a *write pointer* (or `tail`). As is obvious, we write into the buffer at `tail`, and read from `head`. The structure might be defined as follows:

```
typedef struct {
    uint8_t *buffer;
    size_t   buffer_size;
    size_t   head;
    size_t   tail;
    size_t   bytes_avail;
} queue_t;
```

Given this, we can write simple read (`get`) and write (`put`) routines using byte-by-byte accesses:

```
bool put(queue_t *q, uint8_t *data, size_t size) {
    if(q->buffer_size - q->bytes_avail < size){
        return false;
    }
    for(size_t i = 0; i < size; i++){
        q->buffer[(q->tail + i) % q->buffer_size] = data[i];
    }
    q->tail = (q->tail + size) % q->buffer_size;
    q->bytes_avail += size;
    return true;
}

bool get(queue_t *q, uint8_t *data, size_t size) {
    if(q->bytes_avail < size){
        return false;
    }
    for(size_t i = 0; i < size; i++){
        data[i] = q->buffer[(q->head + i) % q->buffer_size];
    }
    q->head = (q->head + size) % q->buffer_size;
    q->bytes_avail -= size;
    return true;
}
```

Note that this `put` routine is identical to the one given above, with the exception that we now check that sufficient space is available before attempting to write. I also have purposefully neglected synchronization logic (which would be absolutely necessary for any real application of the circular buffer).

The byte-by-byte access pattern and modular arithmetic embedded in each iteration of the loop make this code horrendously slow. We can instead implement each read or write operation with two `memcpy` calls, where one handles the portion of a message at the end of the buffer, and the other handles the portion at the beginning if we wrapped around.

```
static inline off_t min(off_t a, off_t b) {
    return a < b ? a : b;
}

bool put(queue_t *q, uint8_t *data, size_t size) {
    if(q->buffer_size - q->bytes_avail < size){
        return false;
    }

    const size_t part1 = min(size, q->buffer_size - q->tail);
    const size_t part2 = size - part1;

    memcpy(q->buffer + q->tail, data,         part1);
    memcpy(q->buffer,           data + part1, part2);

    q->tail = (q->tail + size) % q->buffer_size;
    q->bytes_avail += size;
    return true;
}

bool get(queue_t *q, uint8_t *data, size_t size) {
    if(q->bytes_avail < size){
        return false;
    }
```

```
    const size_t part1 = min(size, q->buffer_size - q->head);
    const size_t part2 = size - part1;

    memcpy(data,         q->buffer + q->head, part1);
    memcpy(data + part1, q->buffer,           part2);

    q->head = (q->head + size) % q->buffer_size;
    q->bytes_avail -= size;
    return true;
}
```

An example usage of this circular buffer would be as follows:

```
int main() {
    queue_t queue;

    queue.buffer      = malloc(128);
    queue.buffer_size = 128;
    queue.head        = 0;
    queue.tail        = 0;
    queue.bytes_avail = 0;

    put(&q, "hello ", 6);
    put(&q, "world\n", 7);

    char s[13];
    get(&q, (uint8_t *) s, 13);

    printf(s); // prints "hello world"
}
```

Our code is simple, and it works fine. But why not make it more complicated?

Enter the Page Table

In the early days of personal computing, computers could essentially only run one program at a time. Whatever program we ran would have full, direct access to the physical memory. It turns out that if we want to run several programs together, they have a tendency to fight over what regions of that memory they want to use. The solution to this conflict is *virtual memory*, under which each program *thinks* it controls *all* of the memory, but in reality the operating system decides who's getting memory from where.

The key to virtual memory is the *page table*, through which the operating system informs the CPU of mappings between *virtual pages* and *physical pages*. A page is a contiguous, aligned region of memory of a (typically) fixed size, say around 4 KiB. When a program wants to access some (virtual) memory, the CPU figures out what page it belongs to then uses the page table to index into the physical memory.

The page table for some Program A might look something like this:

Virtual Page Number	Physical Page Number
0	*null*
1	25
2	12
3	*null*
4	56
…	…

Meanwhile, Program B's could be as follows:

Virtual Page Number	Physical Page Number
0	11
1	*null*
2	92
3	21
4	*null*
…	…

The key point is that there doesn't need to be any rhyme or reason to the way physical pages are assigned to virtual pages. They can be out-of-order with respect to the virtual pages, and some virtual pages don't need physical memory assigned to them at all. Our two programs can use the same virtual page number to refer to different physical pages, and thus they don't need to worry about conflicting with each other's memory.

The details behind figuring out the page numbers, assigning memory, and indexing into the page table are out of the scope of this article, but are unnecessary for understanding what's to come.

A Few Ordinary System Calls, and One That glibc Doesn't Want You to Know About

Before we proceed to exploit the page table in optimizing our circular buffer, let's review some common Linux system calls.

System Call	Description
`int getpagesize()`	Returns the number of bytes in a page of memory. (Technically not always a system call, but close enough.)
`int ftruncate(int fd, off_t length)`	Sets the size of a file to `length` using its file descriptor, `fd`.
`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`	Maps the file described by `fd` into memory and returns a pointer the new memory region. Can be manipulated to do evil through configuration `flags`.

And now, a less common system call that glibc (our friendly userland interface to the kernel) doesn't expose to us:

System Call	Description
`int memfd_create(const char *name, unsigned int flags)`	Returns a file descriptor for an `anonymous file` which exists only in memory.

Interesting! Because glibc doesn't implement it, we will need to write a little bit of wrapper code if we want to use it.

```
int memfd_create(const char *name, unsigned int flags) {
    return syscall(__NR_memfd_create, name, flags);
}
```

Excellent, now we'll be able to call it like any other function. This is going to let us allocate memory and manipulate it like a file. Since it acts like a file, we'll be able to mmap it wherever we want (spoiler alert).

Evil Black Magic Page Table Hacking

Okay, let's get down to business. Let's do what Wikipedia said and make two adjacent pages point to the same memory. We'll need to modify our queue structure and extend its API with a convenient initialization method.

```
typedef struct {
    uint8_t *buffer;
```

```
    size_t   buffer_size;
    int      fd;
    size_t   head;
    size_t   tail;
} queue_t;

bool init(queue *q, size_t size){

    // First, make sure the size is a multiple of the page size
    if(size % getpagesize() != 0){
        return 0;
    }

    // Make an anonymous file and set its size
    q->fd = memfd_create("queue_buffer", 0);
    ftruncate(q->fd, size);

    // Ask mmap for an address at a location where we can put both virtual copies of
    q->buffer = mmap(NULL, 2 * size, PROT_NONE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    // Map the buffer at that address
    mmap(q->buffer, size, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_FIXED, q->fd, 0);

    // Now map it again, in the next virtual page
    mmap(q->buffer + size, size, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_FIXED, q->
```

```
    // Initialize our buffer indices
    q->head = 0;
    q->tail = 0;
}
```

Okay, what's happening? Well, first we call `memfd_create`, which does exactly what it says on the tin. We find some space for two copies of the buffer, and store that as our buffer address. Then we map the virtual file into memory, at the address mmap gave us. After that is where our evil kicks in.

We ask `mmap` to map our file again at the address `size` bytes past its original location. Now, the page table is going to look something like this (assuming `size` is larger than one page):

Virtual Address	Physical Address
…	…
`q->buffer`	anonymous file `q->fd`, offset 0
`q->buffer + getpagesize()`	anonymous file `q->fd`, offset `getpagesize()`
…	…
`q->buffer + size`	anonymous file `q->fd`, offset 0
`q->buffer + size + getpagesize()`	anonymous file `q->fd`, offset `getpagesize()`
…	…

Yes, we have two virtual pages that point to the same physical page, for each page in our buffer. Let's see how this will affect our circular buffer logic:

```
bool put(queue_t *q, uint8_t *data, size_t size) {
    if(q->buffer_size - (q->tail - q->head) < size){
        return false;
    }
    memcpy(&q->buffer[q->tail], data, size);
    q->tail += size;
    return true;
```

```
}

bool get(queue_t *q, uint8_t *data, size_t size) {
    if(q->tail - q->head < size){
        return false;
    }
    memcpy(data, &q->buffer[q->head], size);
    q->head += size;
    if(q->head > q->buffer_size) {
       q->head -= q->buffer_size;
       q->tail -= q->buffer_size;
    }
    return true;
}
```

We allow our `tail` offset to advance into the second virtual memory region, past the bounds of the "real" buffer. However, it still writes to the same physical memory. Similarly, we can read past the bounds of the first region, and by going into the second region we end up reading the same physical memory as if we had manually wrapped around.

Notice that we need to make only one memcpy call instead of two. We check after reading data that the *head* pointer hasn't moved into the second virtual copy of the buffer. If it did, we can decrement both by the buffer size; they will point to the same physical memory, though the virtual addresses will be different. The API remains the same as before.

How Good Is It?

The key difference here is that we can *always* access contiguous blocks of virtual memory, instead of worrying about wrapping around mid-message. We can write a simple micro-benchmark that isolates this behavior as follows:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <sys/time.h>

#define BUFFER_SIZE (1024)
#define MESSAGE_SIZE (32)
#define NUMBER_RUNS (1000000)

static inline double microtime() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return 1e6 * tv.tv_sec + tv.tv_usec;
}

static inline off_t min(off_t a, off_t b) {
    return a < b ? a : b;
}

int main() {
    uint8_t message[MESSAGE_SIZE];
    uint8_t buffer[2 * BUFFER_SIZE];
    size_t offset;

    double slow_start = microtime();
    offset = 0;
    for(int i = 0; i < NUMBER_RUNS; i++){
        const size_t part1 = min(MESSAGE_SIZE, BUFFER_SIZE - offset);
        const size_t part2 = MESSAGE_SIZE - part1;
        memcpy(buffer + offset, message, part1);
```

```
            memcpy(buffer, message + part1, part2);
            offset = (offset + MESSAGE_SIZE) % BUFFER_SIZE;
        }
        double slow_stop = microtime();

        double fast_start = microtime();
        offset = 0;
        for(int i = 0; i < NUMBER_RUNS; i++){
            memcpy(&buffer[offset], message, MESSAGE_SIZE);
            offset = (offset + MESSAGE_SIZE) % BUFFER_SIZE;
        }
        double fast_stop = microtime();

        printf("slow: %f microseconds per write\n", (slow_stop - slow_start) / NUMBER_RUN
        printf("fast: %f microseconds per write\n", (fast_stop - fast_start) / NUMBER_RUN

        return 0;
    }
```

On my i5-6400, I get pretty consistent results, looking something like this:

```
slow: 0.012196 microseconds per write
fast: 0.004024 microseconds per write
```

Note that the single-memcpy code is about three times faster than the double-memcpy code. We have a much larger margin over the naive byte-by-byte copy, which benchmarked at 0.104943 microseconds per write. This micro-benchmark is not wholly representative of the full queue logic (which could be affected by conditions such as page faults and TLB misses), however gives us a good indication that our optimization was worthwhile.

What Just Happened?

Apparently this trick has been known for a while, but is seldom used. Nonetheless, it's an excellent example of taking advantage of a machine's low-level behaviors to optimize software.

If you're interested in seeing my full circular buffer implementation, you can find it on GitHub. The version there is complete with synchronization, error checking, and access at the granularity of arbitrarily-sized messages.