目录

一、背景

1.1 相关数据结构

```
typedef struct _ringbuffer_t ringbuffer_t;
struct _ringbuffer_t
{
    cat_uint8_t      *p_buffer;     /* 放置数据的缓冲区地址 */
    cat_uint32_t      ring_mask;    /* 用于循环头尾索引的掩码，用于代替取余操作 */
    cat_uint32_t      tail_index;   /* 尾部索引 */
    cat_uint32_t      head_index;   /* 头部索引 */
};
```

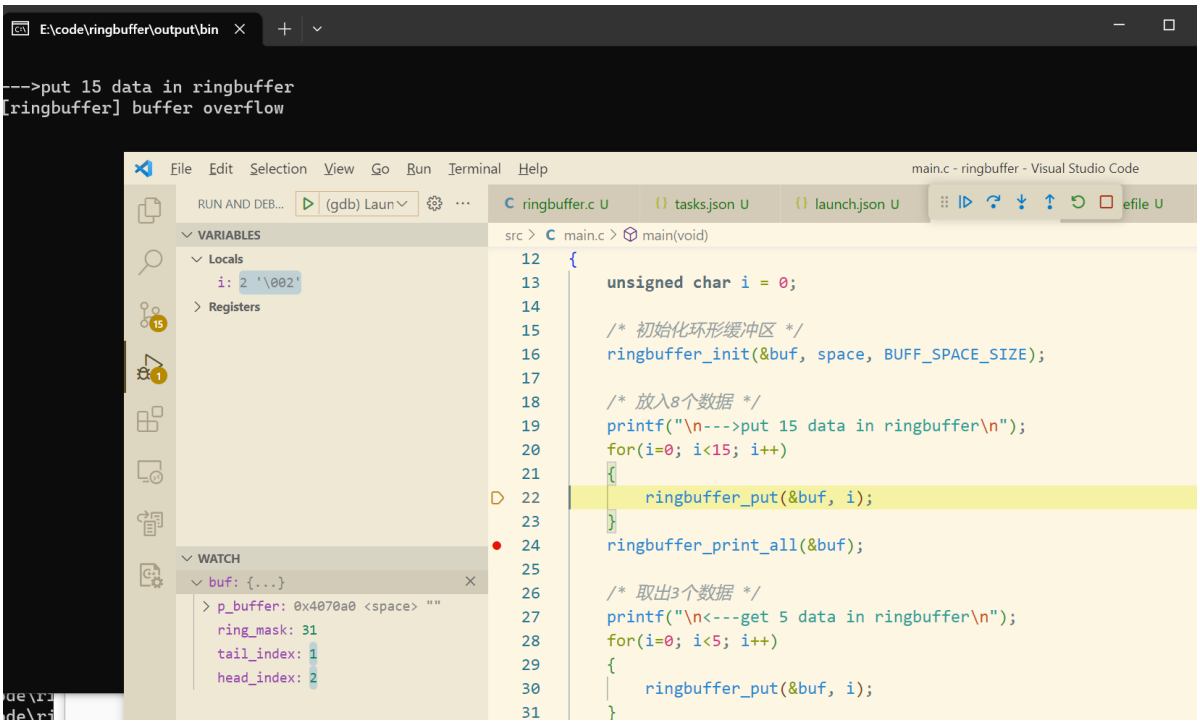
1.2 函数原型

```
static inline cat_uint8_t ringbuffer_is_full(ringbuffer_t *p_ringbuffer)
{
    cat_uint8_t ret;

    ret = (
        (p_ringbuffer->head_index - p_ringbuffer->tail_index) &
(p_ringbuffer->ring_mask) ==
        (p_ringbuffer->ring_mask)
        );

    return ret;
}
```

> note: 使用 ring_mask = buffer_size - 1 来代替取余操作，但是要求 buffer_size 必须为 2^n

二、问题描述

本来缓冲区没有满，计算出来还是满了，如图，只放入了一个数据，放入第二个数据时就提示溢出了



三、问题原因

3.1 c 源码分析

因为 `ringbuffer.c` 中 `ringbuffer_is_full` 函数里

```
ret = (
        (p_ringbuffer->head_index - p_ringbuffer->tail_index) & (p_ringbuffer->ring_mask) ==
        (p_ringbuffer->ring_mask)
        );
```

简写一下

```
(head_index - tail_index) & (ring_mask) == (ring_mask)
```

少了一对括号，因此本来缓冲区没有满，计算出来还是满了

正确的代码应该是

```
ret = (
        ((p_ringbuffer->head_index - p_ringbuffer->tail_index) & (p_ringbuffer->ring_mask)) ==
        (p_ringbuffer->ring_mask)
        );
```

3.2 不同情况的反汇编分析

给出反汇编的区别

3.2.1 没有括号的ringbuffer.o 的 objdump 结果

```
1  static inline cat_uint8_t ringbuffer_is_full(ringbuffer_t *p_ringbuffer)
2  {
3  ...
4      ret = (
5          (p_ringbuffer->head_index - p_ringbuffer->tail_index) &
   (p_ringbuffer->ring_mask) ==
6          /* p_ringbuffer->head_index - p_ringbuffer->tail_index -> %eax寄存器
   */
7     6:   8b 45 08                mov    0x8(%ebp),%eax
8     9:   8b 40 0c                mov    0xc(%eax),%eax
9     c:   89 c2                   mov    %eax,%edx
10    e:   8b 45 08                mov    0x8(%ebp),%eax
11   11:   8b 40 08                mov    0x8(%eax),%eax
12   14:   29 c2                   sub    %eax,%edx
13   16:   89 d0                   mov    %edx,%eax
14     ret = (
15   18:   83 e0 01                and    $0x1,%eax
16   1b:   88 45 ff                mov    %al,-0x1(%ebp)
17         (p_ringbuffer->ring_mask)
18         );
19 ...
```

3.2.1 有括号的ringbuffer.o 的 objdump 结果

```
1  static inline cat_uint8_t ringbuffer_is_full(ringbuffer_t *p_ringbuffer)
2  {
3  ...
4      ret = (
5          ((p_ringbuffer->head_index - p_ringbuffer->tail_index) &
   (p_ringbuffer->ring_mask)) ==
6     6:   8b 45 08                mov    0x8(%ebp),%eax
7     9:   8b 50 0c                mov    0xc(%eax),%edx
8     c:   8b 45 08                mov    0x8(%ebp),%eax
9     f:   8b 40 08                mov    0x8(%eax),%eax
10   12:   29 c2                   sub    %eax,%edx
11   14:   8b 45 08                mov    0x8(%ebp),%eax
12   17:   8b 40 04                mov    0x4(%eax),%eax
13   1a:   21 c2                   and    %eax,%edx
14         (p_ringbuffer->ring_mask)
15   1c:   8b 45 08                mov    0x8(%ebp),%eax
16   1f:   8b 40 04                mov    0x4(%eax),%eax
17         ((p_ringbuffer->head_index - p_ringbuffer->tail_index) &
   (p_ringbuffer->ring_mask)) ==
18   22:   39 c2                   cmp    %eax,%edx
19   24:   0f 94 c0                sete   %al
20     ret = (
21   27:   88 45 ff                mov    %al,-0x1(%ebp)
22         );
23 ...
```

从没有括号的反汇编代码第15行：

```
18: 83 e0 01                and    $0x1,%eax
```

可以看出是将`(p_ringbuffer->head_index - p_ringbuffer->tail_index)`和`(p_ringbuffer->ring_mask) == (p_ringbuffer->ring_mask)`相与了

~~但是奇怪的是 c 语言中按位与(&)的优先级比 == 更高~~

~~我麻子~~

四、总结

~~虽然在 c 语言中 & 比 == 运算符优先级更高~~，但是 & 是右结合，所以如果不加括号会让 == 先处理

作为按位与的 & 比 == 运算符优先级低并且 & 还是右结合

附上优先级表

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	--
	()	圆括号	(表达式）/函数名(形参表)		--
	.	成员选择（对象）	对象.成员名		--
	->	成员选择（指针）	对象指针->成员名		--
2	-	负号 运算符	-表达式	右到左	单目运算符
	~	按位取反运算符	~表达式		
	++	自增运算符	++变量名/变量名++		
	--	自减运算符	--变量名/变量名--		
	*	取值运算符	*指针变量		
	&	取地址运算符	&变量名		
	!	逻辑非运算符	!表达式		
	(类型)	强制类型转换	(数据类型)表达式		--
	sizeof	长度运算符	sizeof(表达式)		--
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		
	%	余数（取模）	整型表达式%整型表达式		
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		
	<	小于	表达式<表达式		
	<=	小于等于	表达式<=表达式		
7	==	等于	表达式==表达式	左到右	双目运算符
	！=	不等于	表达式!= 表达式		
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式1? 表达式2: 表达式3	右到左	三目运算符
	=	赋值运算符	变量=表达式		--
	/=	除后赋值	变量/=表达式		--

14	*=	乘后赋值	变量*=表达式	右到左	--
	%=	取模后赋值	变量%=表达式		--
	+=	加后赋值	变量+=表达式		--
	-=	减后赋值	变量-=表达式		--
	<<=	左移后赋值	变量<<=表达式		--
	>>=	右移后赋值	变量>>=表达式		--
	&=	按位与后赋值	变量&=表达式		--
	^=	按位异或后赋值	变量^=表达式		--
	|=	按位或后赋值	变量|=表达式		--
15	,	逗号运算符	表达式,表达式,...	左到右	--

说明：

同一优先级的运算符，运算次序由结合方向所决定。

简单记就是：！ > 算术运算符 > 关系运算符 > && > || > 赋值运算符