

PA2 实验报告

211240001 潘昕田

2022 年 10 月 28 日

1 实验进度

1. PA2 阶段 1 完成
2. PA2 阶段 2 完成
3. PA2 阶段 3 完成 (选做部分也实现)

2 必答题

2.1 程序是个状态机

2.1.1 画出在 YEMU 上执行的加法程序的状态机

$(0, x, x)(16, 33, 0) \rightarrow (1, 33, x)(16, 33, 0) \rightarrow (2, 33, 33)(16, 33, 0) \rightarrow$
 $(3, 16, 33)(16, 33, 0) \rightarrow (4, 49, 33)(16, 33, 0) \rightarrow (5, 49, 33)(16, 33, 49) \rightarrow$
 $(6, 98, 33)(16, 33, 49)$ (第一个括号标识的是 pc 和寄存器的资料, 第二个括号标识的是 x,y,z 在存储器对应的地址的资料)

2.1.2 通过 RTFSC 理解 YEMU 如何执行一条指令

YEMU 首先会从 pc=0 处的地址读取资料并执行指令, 指令会从 y 所在的地址读取资料并送至寄存器 r0, 接着 pc 自增 1, 读取下一条指令.

根据 pc=1 处的指令, 会将 r0 中的资料送至 r1, 接着 pc 自增 1, 读取下一条指令.

根据 pc=2 处的指令, 指令会从 x 所在的地址读取资料并送至寄存器 r0, 接着 pc 自增 1, 读取下一条指令.

根据 pc=3 处的指令, 指令会将 r0 中的资料和 r1 中的资料相加, 结果送至寄存器 r0, 接着 pc 自增 1, 读取下一条指令.

根据 pc=4 处的指令, 指令会将 r0 中的资料存储至 z 所在的地址, 接着 pc 自增 1, 读取下一条指令.

pc=5 处为 x 的资料, 但也可以视作将 r0 中的资料相加 (2R[r0]) 送至 r0, 接着 pc 自增 1, 读取下一条指令.

pc=6 为 y 的资料, 无法翻译为对应的指令, 根据执行步骤, halt=1, 程序退出. 输出最终 z 的值为 49

2.2 请整理一条指令在 NEMU 中的执行过程

cpu-exec 首先会调用 execute 函数, execute 函数接着会在 for 循环内调用 exec_once 函数, exec_once 首先调用 isa_exec_once.

接着, isa_exec_once 会调用 inst_fetch, 读取 pc 所在地址的资料 (即指令), 同时会将 Decode 结构体中 snpc 值加 4, 接着调用 decode_exec 函数进行译码操作, 得到 dnpc 的值, 之后返回至 exec_once 位置, 根据 dnpc 的值更新 pc 的值

2.3 理解打字小游戏如何运行

打字小游戏首先调用 ioe_init 初始化 ioe, 接着调用 video_init 函数初始化每个字符的表示格式.

接下来进入 while 循环执行游戏的主体内容, 在每一个 while 循环执行中, 程序首先从时间寄存器读取当前的时间并转化为对应的帧数, 接着从上一帧的帧数开始更新游戏, 按照 FPS 和 CPS 的定义可知, 游戏每 6 帧 (严格意义上是当前帧数 mod 6 为 0) 会生成一个新字符 (前提是有空间能够存储, 通过 if(!c->ch) 来确保这一点), 然后判断 c->t 是否大于 0 (大于 0 意味着该字符是 miss, 会在游戏界面底部停留一段时间后再消失), c->t=0 意味着滞留时间结束, 将字符清空, 如果 c->t 一开始即为 0, 则根据 c->v 更新 c->y 的值, 如果 c->y < 0 意味着是成功命中的字符, 将其清空, 反之, 则判断其距离是否大于等于游戏界面的纵轴值, 若大于等于, 意味着该字符 miss, 需要使其在底部停留约 30 帧的时间. 因此 c->t 被赋予 FPS 的值. 同时增加 miss 的次数

随后游戏会从键盘中读取数据, 如果此时未按下按键, 则跳出循环, 若为 ESC 键的键码, 则调用 halt(0) 退出程序, 如果是字母的键码, 则执行

check_hit 来检查是否成功 hit, check_hit 会优先选取相同字符中距离底部最低的字符作为本次 hit 的对象, 然后将其 c->v 的值置为负 (这样着下一次游戏更新时会清除该字符) 并将 hit 值加 1. 若没有匹配到对应的字符, 则判定为错误输入, wrong 值加 1.

之后判断当前帧会先前更新屏幕的帧数的大小, 若当前帧更大, 说明游戏发生了更新, 需要更新 VGA 的相关寄存器, 调用 render 函数刷新屏幕, 然后将 rendered 值置为 current 的值, 以便下一次判断.

2.4 编译与链接

2.4.1 static inline

去掉 static 只保留 inline 不会报错, 去掉 inline 只保留 static 也不会报错, 但如果同时去除则会报出 multiple definition 的错误. 原因在于头文件被多个.c 文件引用, 在编译时首先是.c 分别单独和头文件编译成.o 文件再进一步进行链接, 若不使用 static 关键字会生成多个在不同文件中均可见但符号相同的函数导致多重定义的错误, static 会确保为每个.c 文件仅在该文件中可见的函数. 但是该函数编译器是可以进行 inline 优化, inline 优化会将函数直接内联至调用该函数的函数中, 使得原本的函数并不会被定义, 而是直接将其内容”嵌入”到调用的位置, 因此只有 inline 时编译亦不会报错, 只有 static 和 inline 两个关键完全去除后才会产生重复定义的错误.

2.4.2 volatile static int dummy

1. 在 nemu/build/obj-riscv32-nemu-interpreter 下使用 `grep -r -c 'dummy' ./ * | grep '\.o:[1-9]'` wc -l 可以得到有 34 个 dummy 变量实体.
2. 仍然为 34 个, 原因在于引入 debug.h 的.c 文件的集合为引入 common.h 的.c 文件的子集, 而 dummy 在两处均为定义, 为弱符号, 编译器会随机选择一个并覆盖另一个定义.
3. 编译报错, 此时两个 dummy 因为均进行了初始化, 都是强符号, 因此导致符号冲突, 导致重复定义错误.

2.5 了解 Makefile

敲入 make 指令后, 指令首先找寻当前目录的 Makefile, 给变量 NAME 和 SRC 赋值并引入 abstract-machine 的 Makefile, abstract-machine 的 Make-

file 会根据 ARCH 的参数引入 scripts 文件夹下的 riscv32-nemu.mk, 进一步引入 platform 下的 nemu.mk 和 isa 下的 riscv32.mk 文件.

在 abstract-machine 下的 Makefile 定义了 image 的依赖关系, 执行中该依赖关系会与 nemu.mk 中的 image 的依赖关系相合并, command 的执行根据 nemu.mk 中 image 的 command 定义执行之, 而 nemu.mk 中有 run 规则的描述则 make 指令最终会按照找到的 run 规则生成 nemu-interpretter 并装载 hello 的镜像文件

3 选做题

3.1 立即数背后的故事 (一)

此时读取大于 1 字节的资料将无法使用类型转换来实现, 因为 nemu 是实现在 x86-64 上的一个模拟器, 使用类型转换将采取小端方式, 对大端机而言将是错误, 但单个字节的读取是正确的, 因此若为大端架构, 必须要逐个字节装载读取值, 而不能简单的通过类型转换来实现不同资料的读取.

3.2 立即数背后的故事 (二)

riscv32 指令可以通过 auipc 和 lui 组合来装载 32 位立即数, 考虑到 lui 是按符号扩展至 32 位, 所以可能需根据第 11 位 (0 至 11 位) 的 0,1 情况判断是否需要在第 12 位补充加上 1 以抵消符号扩展带来的影响.

3.3 指令名对照

根据 RISC-V 官网提供的 Volume 1, Unprivileged Spec v. 20191213 的第 25 章 RISC-V Assembly Programmer's Handbook(非 draft 版本, draft 版本无此章节) 可以查询伪指令和汇编指令的对应关系

3.4 这又能怎么样呢?

进行 API 的抽象可以节省开发效率, 不需要每一份程序都编写一份全新的函数, 同时, 当发现某个架构下的 API 实现有 Bug, 只要修改一份代码即可解决问题, 而不必去把 n 个程序里的代码去修改, 极大地提升开发效率和修正错误的能力.

3.5 stdarg 是如何实现的?

在 i386 架构上, 函数上所有的参数均通过栈来传递, 因此只需要知道可变参数前一个参数的地址即可算出可变参数的起始位置并逐步更新地址并读取参数.

然而在 x86-64 架构上, 函数会将前 6 个整数型参数和前 8 个浮点型参数通过寄存器传递, 此时无法使用先前的做法来实现参数地址的计算, 解决方法通过 STFW 可以得知. 编译器会将 variadic function(具体地, 是在 function 中使用了 `va_start` 的 function) 的寄存器值在函数的栈上开辟一个区域存储, 并存储一个浮点寄存器的地址偏移值和通用寄存器的地址偏移值, 同样是根据可变参数前一个参数决定偏移值, 但是此时宏命令会先判断此类型变量在其对应的寄存器保留值区域 (`register_saved_area`) 的偏移是否溢出, 若溢出, 转至堆栈传参的区域去获取参数, 否则在 `register_saved_area` 中获取参数的值.

因此当自己实现 `stdarg` 时, 需要维护一个结构体来存储普通的栈上参数和寄存器传递的参数, 同时存储两个偏移值 (浮点型和整型寄存器在栈上的存储位置), 通过偏移值是否溢出来判断参数的地址. 从而实现 variadic function 参数的读取

3.6 消失的符号

宏命令会在预编译时展开, 局部变量存储在栈上, 它们并不会在重定位过程起作用. 而符号是用于在重定位过程中改写函数或者外部全局变量地址, 因此只有全局变量和函数会有 `symbol`.

3.7 寻找"Hello World!"

在 `.rodata` 段中, 因为 `strtab`(字符串表) 存储的是符号的字符串, 而 "Hello World!" 是硬编码到程序中的数据, 而且字符串是只读的, 不可对其进行修改, 因此在 `.rodata` 段中.

3.8 不匹配的函数调用和返回

可以看到 `f2, f3` 均有 `sw` 指令而 `f0, f1` 没有, 说明 `f0, f1` 并没有在栈上存储局部变量, 进一步查询可知 `f0, f1` 使用了尾递归有优化, 会返回至第一次调用该函数的位置而不是逐层返回, 因此出现了不匹配的情况.

3.9 冗余的符号表

删除可执行文件的符号表并不会影响程序的执行,但是删除目标文件的符号表会导致编译错误,原因在于符号表是用于进行重定位,在可执行文件中没有作用.

3.10 如何生成 native 的可执行文件

根据 CFLAGS 中的-D 选项,以及 make 中传入的 ARCH 参数,gcc 会生成不同的宏,从而编译不同的头文件等具有对应宏定义的内容,从而生成不同平台上的编译产物.

3.11 奇怪的错误码

RTFM 可得

图 1: Make 关于 exit code 的文档

The exit status of make is always one of three values:

0

The exit status is zero if make is successful.

2

The exit status is two if make encounters any errors. It will print messages describing the particular errors.

1

The exit status is one if you use the '-q' flag and make determines that some target is not already up to date. See [Instead of Executing Recipes](#).

在 Makefile 编译过程中开启了-q 选项,因此当遇到编译失败导致部分文件未能更新至最新状态时,make 会以 1 状态退出.

3.12 这是如何实现的?

阅读代码可得 klib 中所有的函数只有在非 NATIVE 环境下或者 __NATIVE_USE_KLIB__ 的情况下才会编译,成为强符号,此时将不会调用 glibc 来链接而是直接使用 klib 实现的库函数.

图 2: klib 中库函数编译的条件

```
#if !defined(__ISA_NATIVE__) || defined(__NATIVE_USE_KLIB__)
```

3.13 理解 volatile 关键字

使用 volatile 编译时

图 3: 使用 volatile 时的代码及反汇编结果

```
1 void fun() {
2     extern unsigned char _end;
3     volatile unsigned char *p = &_amp;_end;
4     *p = 0;
5     while(*p != 0xff);
6     *p = 0x33;
7     *p = 0x34;
8     *p = 0x86;
9 }
10
11 int main(){
12     return 0;
13 }
14 }
```

```
0000000000001140 <fun>:
1140:    f3 0f 1e fa    endbr64
1144:    c6 05 cd 2e 00 00 00    movb    $0x0,0x2ecd(%rip)    # 4018 <_end>
114b:    48 8d 15 c6 2e 00 00    lea     0x2ec6(%rip),%rdx    # 4018 <_end>
1152:    66 0f 1f 44 00 00    nopw    0x0(%rax,%rax,1)
1158:    0f b6 02    movzbl  (%rdx),%eax
115b:    3c ff    cmp     $0xff,%al
115d:    75 f9    jne     1158 <fun+0x18>
115f:    c6 05 b2 2e 00 00 33    movb    $0x33,0x2eb2(%rip)    # 4018 <_end>
1166:    c6 05 ab 2e 00 00 34    movb    $0x34,0x2eab(%rip)    # 4018 <_end>
116d:    c6 05 a4 2e 00 00 86    movb    $0x86,0x2ea4(%rip)    # 4018 <_end>
1174:    c3    ret
```

不使用 volatile 编译时

图 4: 不使用 volatile 时的代码及反汇编结果

```
1 void fun() {
2     extern unsigned char _end;
3     unsigned char *p = &_end;
4     *p = 0;
5     while(*p != 0xff);
6     *p = 0x33;
7     *p = 0x34;
8     *p = 0x86;
9 }
10
11 int main(){
12     return 0;
13 }
14
```

```
00000000000001140 <fun>:
1140: f3 0f 1e fa      endbr64
1144: c6 05 cd 2e 00 00 00 movb $0x0,0x2ecd(%rip) # 4018 <_end>
114b: eb fe          jmp 114b <fun+0xb>
```

可以看到使用 volatile 时不会对 p 进行任何优化, 而不使用时则会直接优化为一个死循环. 这是因为使用 volatile 后该变量的内存被视作随时可能发生改变, 因此对其不会作任何优化 (包括删除相关内容或者调整执行顺序).

volatile 关键字是为了避免优化诸如硬件读取 (读和写的时间和位置可能不一致, 硬件更新不一定会在函数中得到体现) 的函数, 例如在上述代码中若 p 指向的是设备寄存器的内容, 即使在 while 循环前对其赋予 0 值, 从函数本身的角度确实不可能跳出 while 循环, 但此时设备却可能更新了相关值, 使得在进行循环条件判断不满足而退出, 继而可以更新并退出函数. 显然, 优化而得的代码并不是我们所需要的结果, 因此使用 volatile 可以使得和 I/O 交互的变量能够正确执行其逻辑而不会因优化导致错误.

同时 volatile 也可以用于多线程程序, 当某个内存区域可能被多个线程修改时, 对引用该内存区域的变量而言, 如果没有 volatile 也可能会编出错误的逻辑.

p.s. 也考虑过是否能够通过不开启优化来避免, 但这样会造成不必要的效率降低 (除了 volatile 声明的变量外其实是可以优化的, 而且 volatile 声明的变量在程序中占比并不大). 同时阅读过关于 Undefined Behavior 的相关论文, 发现即使是 O0 也会进行一定的优化.

3.14 如何检测多个键同时被按下？

按键按下时会发送通码，我们可以据此记录下按键目前的状态，如果接下来在该按键断码发送前接收到其它键的通码，即可判断是否同时按下，为了更好地体现同时的特征，可以在某一个按键优先侦测到通码后设置很短的时间阈值，在阈值内若未收到断码且收到另一个键的通码，即视为同时按下。

3.15 神奇的调色板

可以通过调整调色板的亮度来实现之。

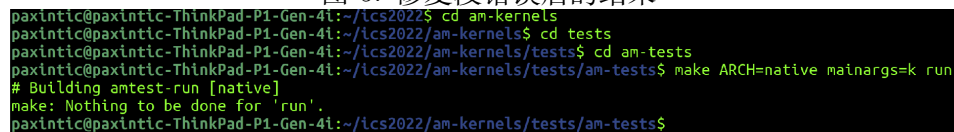
4 实验心得

本次实验的经历颇有些坎坷，大概在 pa1 完成两周后的周末，已经实现 pa2.1 所有指令的执行并且测试 cpu-tests 都通过时，准备着手处理 batch 模式的问题，结果当时没能理解 Makefile 的组织结构，错误地以为打开 batch 模式是要使用 CONFIG_TARGET_AM 这个宏，但看到 menuconfig 里面的内容发现提示不可以打开生成该宏的选项。结果当时以为是可以通过其它手段打开之（现在回过头来是实在不能理解当时自己的想法）。然后在 nemu 的 configs 里面找到了 riscv32-nemu-am_defconfig 文件，发现里面打开了 CONFIG_TARGET_AM，然后作死把它执行了一遍，然后。

nemu 就彻底废了。

生成后由于地址位置等一系列配置发生了变化，导致 nemu 在编译后直接因为地址溢出崩溃，然后又尝试去 menuconfig 里面重新调整，结果发现不会再触发段错误了，但是...

图 5: 修复段错误后的结果



```
paxintic@paxintic-ThinkPad-P1-Gen-41:~/ics2022$ cd am-kernels
paxintic@paxintic-ThinkPad-P1-Gen-41:~/ics2022/am-kernels$ cd tests
paxintic@paxintic-ThinkPad-P1-Gen-41:~/ics2022/am-kernels/tests$ cd am-tests
paxintic@paxintic-ThinkPad-P1-Gen-41:~/ics2022/am-kernels/tests/am-tests$ make ARCH=native mainargs=k run
# Building amtest-run [native]
make: Nothing to be done for 'run'.
paxintic@paxintic-ThinkPad-P1-Gen-41:~/ics2022/am-kernels/tests/am-tests$
```

结果就是之后所有的 make 都无法执行了。

后来从下午一直排查错误至晚上，最终发现自己实在无法解决 Bug(完全不清楚错误的位置)。后来甚至发现 riscv32-nemu-am_defconfig 的内容也发生了变化。绝望之下，当时甚至觉得可能要从 pa0 重新来过（甚至已经下了新的 nemu 准备重开）。后来抱着赌一把的心态，准备回到 pa1 的分支去看看

pa1 是否有错误, 结果发现 pa1 分支的执行没有任何问题, 执行 am-kernels 里面的 cpu-tests 也只是报出没有指令的提示而不是根本无法执行. 于是乎, 找到希望后, 就决定回到 pa1 重开 pa2, 所幸当时只完成了 pa2.1 的部分, 把指令执行内实现的部分先备份, 然后删除 pa2 分支重新创立后在把内容复制回去. 然而, 这样做的代价便是将 pa2 先前所有的 commit 的记录销毁了. 统计了一下大约有 200 多个记录 (因为实现了一部分指令就会去测试, 因此有很多次编译和运行的记录). 说实话删除还是十分纠结的, 但是对于当时的我而言, 已经是不幸中的万幸了.

图 6: 删除原 pa2 分支前 pa2 分支的记录

```
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/ics2022$
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/ics2022$ git checkout pa2
Switched to branch 'pa2'
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/ics2022$ git log
commit f6a468740712cb20c065f15814f0843d128a726d (HEAD -> pa2)
Author: tracer-ics2022 <tracer@njuics.org>
Date: Sat Sep 24 18:57:45 2022 +0800

    > run NEMU
    211240001 潘昕田
    Linux paxintic-ThinkPad-P1-Gen-4i 5.15.0-25-generic #25-Ubuntu SMP Wed Mar 30 15:54:22 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
    18:57:45 up 3:38, 1 user, load average: 0.28, 0.38, 0.45

commit 57e803dbbcfb138d75abd4d3f17750bcae060baf
Author: tracer-ics2022 <tracer@njuics.org>
Date: Sat Sep 24 18:57:44 2022 +0800

    > compile NEMU
    211240001 潘昕田
    Linux paxintic-ThinkPad-P1-Gen-4i 5.15.0-25-generic #25-Ubuntu SMP Wed Mar 30 15:54:22 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
    18:57:44 up 3:38, 1 user, load average: 0.28, 0.38, 0.45

commit 99a1a033ab99c43b9efcdabebe46836f77e0a220
Author: 211240001-Pan Xintian <iamxintianpan@gmail.com>
Date: Sat Sep 24 18:55:35 2022 +0800

    NJU-ProjectN/am-kernels ics2021 initialized without tracing

    NJU-ProjectN/am-kernels adc316af6e482e6444a9bd68bafc3a57e2cafdbc cpu-tests,Makefile: support target 'gdb'

commit b6cab99fd9e348ddf06a5aee356df12d25cb7d46
Author: tracer-ics2022 <tracer@njuics.org>
Date: Sat Sep 24 18:51:14 2022 +0800

    > run NEMU
    211240001 潘昕田
    Linux paxintic-ThinkPad-P1-Gen-4i 5.15.0-25-generic #25-Ubuntu SMP Wed Mar 30 15:54:22 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
    18:51:14 up 3:32, 1 user, load average: 0.52, 0.49, 0.51

commit 07c457ce43333c89c16b3b3de32c10f208356cb1
Author: tracer-ics2022 <tracer@njuics.org>
Date: Sat Sep 24 18:51:14 2022 +0800

    > compile NEMU
    211240001 潘昕田
    Linux paxintic-ThinkPad-P1-Gen-4i 5.15.0-25-generic #25-Ubuntu SMP Wed Mar 30 15:54:22 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
    18:51:14 up 3:32, 1 user, load average: 0.52, 0.49, 0.51
```

不过由于原先 pa1 的结束分支”before starting pa2” 因为后来测试 pa1 的缘故不再为其结束分支, 当时考虑是否删去多出的分支. 后来转念一想, 保留这个记录正好可以表明出过 Bug(不然谁闲得无聊会 commit 两个”before starting pa2”). 当然, 为了强化这个惨痛的经历, 我还加上了这样一段 commit 记录.

图 7: pa1 的分支记录

```
1 problems happens in pa2, reset to pa1(only pa2.1 finshed)
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 #
5 # On branch pa1
```

惨痛的 Debug 经历, 加上那天眼镜又散架了 (印象中应该是周六). 之后周日一天都在阴影中度过, 连 Linux 都不敢打开了.

后来重新审视 Makefile 和 nemu 代码, 细细的品读后, 才发现实际上是需要在 abstract-machine 下面的 nemu.mk 里面的 NEMU_FLAGS 加上 -batch(或者 -b), 通过向 nemu 传递命令行参数来实现 batch 模式的 (NEMU_FLAGS 的内容在编译时会被当作命令行参数传入).

如此惨烈的教训使得我更加意识到 RTFSC 的重要性, 不仅要了解源代码说了什么, 更需要透彻地理解其运作的详细流程, 也因此, 在 pa2.3 中, 我花了大量的时间去理解设备寄存器读取写入的方式, 以及其回调函数的呼叫方式. 因此实际实现时几乎没有遇到错误, 只有键盘读入时出过错 (当时误以为 keydown 状态是在调用时就已经提供了, 实际上是要自己判断是通码还是断码来决定, 结果导致一开始使用 inl 直接报错 (因为通码的数值不在键盘码范围内), 后来仔细排查才发现需要自行判断断码和通码来判断 keydown), 以及绘图时二维的像素转为一维时报错 (因为当时纵轴的遍历因为手误错误采用了横轴的像素个数遍历 (在原本 am-tests 没有错误是因为它是按照正方形的块来更新的, 然而 kernels 里面不是)). 不过由于已经通读过源码, 因此错误排查的速度非常快, 也没有发生之前的惨剧.

pa2 可谓给了我一个大教训, 让我意识到不彻底贯彻 RTFSC 的精神会造成多么巨大的惨剧, 也让我明白与其拿到任务直接开始作业, 不如先多花点时间了解给予你的框架的逻辑, 并且了解的越详细越好, 这也能让你在短时间内写出高质量、少 BUG、鲁棒性好的代码. 不过幸运的是这发生在 pa2 的开头, 而不是 pa3 或者 pa4, 不然就可能真的要走到穷途末路. 在本次实验报告中如此详细写下自己的心得和当时的心路历程也是为了进一步的警示自己, 不要忘了这次教训.

毋忘之!!!

p.s. 写完 pa2 必做部分后又去实现了选做部分, 音乐能够顺利的播放, 但是中间偶尔会夹杂一些杂音 (并不是每一次都会触发), 初步分析可能和队列时间以及回调函数调用有关, 队列维护操作难度较大 (count 能读取占了多少字节但没法读出对首和对尾), 若需要能够完成正确地读和写必须保证

对首恒在数组开头出, 这就导致每次调用回调函数后必须将其剩余部分移至开头, 但由于回调函数有可能与写函数同时调用导致一定的冲突. 不过个人感觉还有可能是 nemu 上执行速度偏慢导致的.