PA2 实验报告

191240046 孙博文

完成情况：完成所有必做题，通过Online-Judging的所有样例。

NEMU的跑分：685分（这是在PA4还没有开启分页的时候跑的分数，不知道为什么比PA2的时候快了不少。）

平台：Intel i5-8265U 4C8T 1.6GHz@3.9GHz 插电

```
[src/device/io/mmio.c,13,add_mmio_map] Add mmio map 'vmem' at [0xa0000000, 0xa00752ff]
[src/device/io/port-io.c,15,add_pio_map] Add port-io map 'keyboard' at [0x00000060, 0x00000063]
[src/device/io/mmio.c,13,add_mmio_map] Add mmio map 'keyboard' at [0xa1000060, 0xa1000063]
[src/device/io/port-io.c,15,add_pio_map] Add port-io map 'audio' at [0x00000200, 0x00000217]
[src/device/io/mmio.c,13,add_mmio_map] Add mmio map 'audio' at [0xa1000200, 0xa1000217]
[src/device/io/mmio.c,13,add_mmio_map] Add mmio map 'audio-sbuf' at [0xa0800000, 0xa080ffff]
======= Running MicroBench [input *ref*] =======
[qsort] Quick sort: * Passed.
  min time: 472 ms [1083]
[queen] Queen placement: * Passed.
  min time: 569 ms [827]
[bf] Brainf**k interpreter: * Passed.
  min time: 2974 ms [795]
[fib] Fibonacci number: * Passed.
  min time: 6615 ms [428]
[sieve] Eratosthenes sieve: * Passed.
  min time: 5843 ms [673]
[15pz] A* 15-puzzle search: * Passed.
  min time: 1275 ms [351]
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 1088 ms [1000]
[lzip] Lzip compression: * Passed.
  min time: 1540 ms [493]
[ssort] Suffix sort: * Passed.
  min time: 481 ms [936]
[md5] MD5 digest: * Passed.
  min time: 6369 ms [270]
================================================
MicroBench PASS        685 Marks
                 vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 32151 ms
[src/monitor/cpu-exec.c,116,cpu_exec] nemu: HIT GOOD TRAP at pc = 0x0010362c


[src/monitor/cpu-exec.c,39,monitor_statistic] total guest instructions = 1787454930
[src/monitor/cpu-exec.c,40,monitor_statistic] host time spent = 32152 ms
[src/monitor/cpu-exec.c,41,monitor_statistic] simulation frequency = 55593895 instr/s
```

必答题

程序是个状态机 理解YEMU的执行过程。

- 主函数中的循环不断调用`exec_once`函数执行一条指令
- 每执行完一条指令之后判断当前的执行状态
- 取指令：将变量pc作为指令指针寄存器，以pc的值在数组M，即模拟的内存数组中寻址，获得当前需要执行的指令。
- 译码：取值后，根据`union inst_t`定义的指令格式，将指令分为操作码域和操作数域，通过一个跳转表，对操作码域进行解码，然后在各自的情况中按opcode决定通过`DECODE_R`或`DECODE_M`对操作数解码。

- 执行：按照不同的指令形式来执行指令。
- 更新pc

RTFSC 整理一条指令在NEMU中的执行过程。

- `isa_exec_once`：
  - 创建一个`DecodeExecState`状态结构体。设置跳转为false，将当前的EIP设置为顺序取指指针的值（`s.seq_pc`）， 用于在执行状态中更新。
  - `fetch_decode_exec(&s)`
    - 从`s.seq_pc`取一字节作为当前的opcode
    - 是一个跳转表的结构。通过宏定义的展开，以此执行各自的`set_width\DHelper\EHelper`函数。部分指令如`0x0f`开头的，进入下一个跳转表，并执行相同的操作。如果opcode为`0x66`，说明是16位形式的指令。设置`s->isa.is_operand_size_16 = true`，再取一个字节。
      - 译码：

        将目的操作数取到`id_dest`中，将源操作数取到`id_src`中，并设置操作数类型。若指令有ModRMbyte，则再取一字节，根据ModRM的各位域判断操作数类型
      - 执行：通过rtl指令，将指令分解成各个执行部分执行。对于运算指令，需要根据结果更新eflags寄存器。
  - `update_pc(&s)`
    - 若在指令执行的过程中，`s.is_jmp`被设置为true，则将`cpu.pc`设置为`jmp_pc`。否则设置为`seq_pc`。

程序如何运行 理解打字小游戏如何运行

- 首先通过`ioe_init\video_init`初始化io设备。
- 若检测到timer和输入设备不存在则退出。
- 外层循环每循环一次就根据当前时间和FPS决定更新游戏逻辑的次数。
- 用一个for循环来更新游戏逻辑。
- 一个`while(1)`循环，从`AM_INPUT_KEYBRD`中读取当前的输入状态。
  - 输入`KEY_NONE`：直接跳出循环。
  - 输入ESC：停止游戏。
  - 输入其他字符：再通过`Lut[ev.keycode]`判断当前的输入是否是字母。然后`check_hit`，判断字母是否在屏幕上。
- 如果现在的游戏逻辑领先于已经渲染的游戏逻辑，则渲染一帧。
- 其中，每个字母都是一个struct，包含了ASCII码， 坐标，速度，以及生成的时间
  - 生成新字母，`new_char()`，如果字母的ASCII码为0， 则随机生成一个字母，随机指定x坐标，将y坐标和生成时间设置为0。
  - 游戏逻辑的更新：更新每一个字母的状态。如果ASCII码为0或生成时间为0则不更新。否则根据速度更新字母的y坐标。如果字母的y坐标+字母的高度>屏幕高度，则判断为miss。
- 画面的渲染：
  - 渲染背景。
  - 遍历chars数组中的所有字符，如果存在这个字符则将这个字符输出到屏幕上的x, y位置。

编译与链接

- 去掉`static`：出现中定义错误，因为它被多个头文件多次引用了。再去掉`inline`也是相同的结果。
- 去掉`inline`：Werror=`unused-function`。因为静态函数被声明了但是没有定义。

- 重新编译后的NEMU含有27个`dummy`变量的实体。用`readelf -syms`，再在vim中统计。
- 再添加后，重新编译，仍含27个`dummy`实体。因为此时头文件中进行的是声明。
- 出现了重定义错误。因为对变量的赋值视为强定义，两次强定义被同一个头文件包含了。

了解Makefile

Makefile的工作方式：

- 默认目标为`app`， 生成最终的可执行文件。它是一个pseudo goal，不会生成文件名为app的可执行文件
- 目标app依赖于`($BINARY)`目标，它会生成一个以`($BINARY)`为文件名的可执行文件
- `($BINARY)`目标依赖于`($OBJS)`目标，进一步依赖于`src/%.c`，通过搜索src目录下的所有.c文件，获得所有的可重定位目标文件。gcc 的`-I ($INCLUDES)`选项指定了引用的头文件的位置为includes/
- 生成所有可重定位目标文件后，链接为最终的可执行二进制文件。