

华中科技大学

2018

系统能力综合训练 课程设计报告

题 目:	X86 模拟器设计
专 业:	计算机科学与技术
班 级:	CS1501
学 号:	U201514479
姓 名:	张一林
电 话:	15927107067
邮 件:	2322162863@qq. com
完成日期:	2019-1-4 周五上午



计算机科学与技术学院

华中科技大学课程设计报告

目 录

1	课程设计概述.....	3
1.1	课设背景	3
1.2	实验目的	3
1.3	实验环境	3
1.4	设计任务	3
2	PA1 简易调试器.....	4
2.1	简易调试器	4
2.2	表达式求值	5
2.3	监视点	8
2.4	必答题	9
3	PA2 冯诺依曼计算机.....	11
3.1	第一个 C 程序	11
3.2	丰富指令集	11
3.3	I/O 指令	13
3.4	必答题	15
4	PA3 批处理系统.....	16
4.1	中断与上下文.....	16
4.2	系统调用	17
4.3	文件系统	19
4.4	批处理系统	20
4.5	必答题	22
5	设计总结与心得	24
5.1	课设总结	24

华中科技大学课程设计报告

5.2	课设心得	24
-----	------------	----

1 课程设计概述

1.1 课设背景

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统。ICS-PA 将提出 x86 架构的一个教学版子集 n86, 指导学生实现一个功能完备的 n86 模拟器 NEMU(NJU EMUlator), 最终在 NEMU 上运行游戏"仙剑奇侠传", 来让学生探究"程序在计算机上运行"的基本原理。NEMU 受到了 QEMU 的启发, 并去除了大量与课程内容差异较大的部分, 通过设计简易调试器, 实现 x86 指令的译码与执行过程, 实现时钟, 键盘, VGA 等输入输出设备, 搭建简易的文件系统以及批处理系统, 实现中断与系统调用, 最终能够在 NEMU 上运行游戏《仙剑奇侠传》。

1.2 实验目的

亲自实现一个完整的计算机系统, 并在其上运行真实的程序, 明白系统栈每一个层次之间的关系, 对“程序如何在计算机上运行”产生深刻的认识。

1.3 实验环境

CPU 架构: x64

操作系统: GNU/Linux

编译器: GCC

编程语言: C 语言

1.4 设计任务

PA1: 简易调试器

PA2: 冯诺依曼计算机系统

PA3: 批处理系统

PA4: 分时多任务

PA5: 程序性能优化

2 PA1 简易调试器

2.1 简易调试器

2.1.1 实现正确的寄存器结构体

在这一步中，主要思路是将/nemu/include/cpu/reg.h 中的结构体 CPU_state 正确实现，以求通过 init_monitor()中的 reg_test()函数。

通过观察可以得知，应该使 gpr 数组与八个通用寄存器变量描述的是同一块内存地址，通过使用匿名联合和匿名结构得以实现。

正确实现后，在 nemu 目录下执行 make run，可以看到(nemu)输入提示符。

```
void@ubuntu:~/ics2018/nemu$ make run
./build/nemu -l ./build/nemu-log.txt -d /home/void/ics2018/nemu/tools/qemu-diff/build/qemu-so
[src/monitor/monitor.c,55,load_default_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c,29,welcome] Debug: ON
[src/monitor/monitor.c,32,welcome] If debug mode is on, A log file will be generated to record every
instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn
it off in include/common.h.
[src/monitor/monitor.c,37,welcome] Build time: 17:44:43, Jan 7 2019
Welcome to NEMU!
For help, type "help"
(nemu)
```

图 2-1nemu 输入提示符

2.1.2 实现基本的调试命令

单步执行

单步执行对应着命令 si N，其中 N 缺省时默认值为 1.通过阅读框架代码，可以知道 cpu_exec()函数既是指令执行函数，其参数是执行的指令条数，因此只需要在 cmd_table 中添加 cmd_si 函数指针以及对应的描述，并在 cmd_si 中使用 strtok 将指令参数取出，再利用 atoi 将字符串转化为整型传入 cpu_exec 即可。

```
(nemu) si 5
100000: b8 34 12 00 00
(nemu) si 5
100005: b9 27 00 10 00
10000a: 89 01
10000c: 66 c7 41 04 01 00
100012: bb 02 00 00 00
100017: 66 c7 84 99 00 e0 ff ff 01 00
(nemu)
```

图 2-2 单步执行

打印寄存器

打印寄存器对应命令 `info r`，具体操作同上，只需在最终的 `cmd_r` 中将全局结构变量 `cpu` 中的各个寄存器打印出来即可。

```
(nemu) info r
eax : 0x00001234 ,esp : 0x3bdfbf2f
ecx : 0x00100027 ,ebp : 0x2e2787f5
edx : 0x6d835c86 ,esi : 0x13e5ee70
ebx : 0x00000002 ,edi : 0x29cea42c
eip : 0x00100021
```

图 2-3 打印寄存器

扫描内存

扫描内存对应命令 `x N addr`，可以利用函数 `guest_to_host` 将传入的 `addr` 转换为实际的内存指针，再利用 `for` 循环将 `N` 组数据打印出来即可。

```
(nemu) x 10 0x100000
0x100000
1048576 100000
0x001234b8
0x0027b900
0x01890010
0x0441c766
0x02bb0001
0x66000000
0x009984c7
0x01ffffe0
0x0000b800
0x34d60000
```

图 2-4 扫描内存

2.2 表达式求值

表达式计算

在此环节中只需要实现基本的操作数和运算符：十进制正整数，加减乘除，等于，括号，空格即可。

首先需要在 `/nemu/src/monitor/debug/expr.c` 中的 `rules` 中添加正确的正则匹配规则，之后在 `make_token` 中会通过循环来逐个尝试匹配 `rules` 中的规则。

其次，我们需要完善 `make_token` 函数，当成功匹配之后将匹配到的字符串封装成 `Token` 放入 `tokens` 数组，因为空格是无意义的，所以匹配到空格时我们可以直接丢弃。

之后，我们需要根据已经获得的 `tokens` 数组在 `eval` 函数中进行求值，求值的过

程我们可以按照表达式计算的归纳定义来递归地进行。

```
<expr> ::= <number>      # 一个数是表达式
| "(" <expr> ")"          # 在表达式两边加个括号也是表达式
| <expr> "+" <expr>        # 两个表达式相加也是表达式
| <expr> "-" <expr>        # 接下来你全懂了
| <expr> "*" <expr>
| <expr> "/" <expr>
```

图 2-5 递归求值 BNF

在这个过程中，我们需要首先实现 `check_parentheses` 函数来判断表达式的两端是否是一对匹配的括号，如同上图中的第二条规则，若是一堆匹配的括号，即可将括号消去，即执行 `eval(p + 1, q - 1)`。而当前两条规则不符合时，我们需要找到当前表达式中得主运算符，也就是实现函数 `found_main_token`。

在寻找主运算符时，我们可以从右往左进行扫描，遇到的第一个括号外的加减符号既是主运算符，当没有加减运算符时，最先遇到的乘除符号既是著运算符。

找到了主运算符后，我们就可以以主运算符为核心，将表达式分裂成两半，分别对左右两边递归求值，最后根据主运算符将两个值合并，即可得到表达式最终的值。

随机表达式生成器

因为我们将来是要使用自己实现的表达式求值功能来帮助进行后续的调试的，这意味着程序设计课上那种“代码随便测试一下就交上去然后就可以撒手不管”的日子已经一去不复返了。测试需要测试用例，通过越多测试，我们就会对代码越有信心。但如果自己来设计测试用例是不现实的，有没有一种方法来自动产生测试用例呢？

一种常用的方法是随机测试。首先我们需要来思考如何随机生成一个合法的表达式。事实上，表达式生成比表达式求值要容易得多。

```
void gen_rand_expr() {  
    switch (choose(3)) {  
        case 0: gen_num(); break;  
        case 1: gen('('); gen_rand_expr(); gen(')'); break;  
        default: gen_rand_expr(); gen_rand_op(); gen_rand_expr(); break;  
    }  
}
```

图 2-6 随机表达式生成范式

只要遵循着这样的归纳规则，我们就可以很容易地实现随机表达式生成函数，在这之后我们可以将生成的表达式字符串利用 `sprintf` 写入到一个 c 文件中，并利用编译器自动求值，并将求得的值写入一个文件中。

```
static char *code_format =  
"#include <stdio.h>\n"  
"int main() { "  
"    unsigned result = %s; "  
"    printf(\"%%u\", result); "  
"    return 0; "  
"}";
```

图 2-7 借助 c 程序求值

当成功实现以上步骤后，我们就能够在 `main` 函数中读入随机表达式和值来进行验证。

```
6 6  
1 ( 1)  
15 ( ( 5 / 7)) + 8 + 7  
1 ( ( 1))  
8 8  
2 2  
1 1  
2 2  
2 ( ( 1) + 1)  
4 4
```

图 2-8 随机生成示例

```
(nemu) p (100 + 12)/24*(32 - 1)  
Expression Value is dec:124, hex:0x0000007c
```

图 2-9 表达式求值

2.3 监视点

扩展表达式求值的功能

在这个步骤中,我们首先要将之前的简易求值进行扩充,增加对十六进制,负数,寄存器,解引用,各种逻辑运算符的支持。

我们首先要在 `rules` 中增加规则,在这里需要注意规则书写的顺序,例如将十进制整数的匹配规则放在十六进制之前会导致将 `0x` 中的 `0` 匹配为十进制整数 `0`,从而发生错误。

此外,我们暂时可以不区分负号与减号,解引用与乘号在匹配上的区别。当在 `make_token` 中匹配到-或*时,要根据上一个 `Token` 来进行判断,若上一个 `Token` 是运算符就意味着该运算符是符号或解引用而非减号或乘号。

在 `eval` 中同样需要进行扩展,增加如下的一些规则,具体实现同 2.2 中的操作。

```
<expr> ::= <decimal-number>
| <hexadecimal-number>      # 以"0x"开头
| <reg_name>                  # 以"$"开头
| "(" <expr> ")"
| <expr> "+" <expr>
| <expr> "-" <expr>
| <expr> "*" <expr>
| <expr> "/" <expr>
| <expr> "==" <expr>
| <expr> "!=" <expr>
| <expr> "&&" <expr>
| "*" <expr>                  # 指针解引用
```

图 2-10 拓展后的求值范式

在这些都实现了之后,还需要对寻找主运算符函数 `found_main_token` 进行一定的处理,就我个人而言是增加了运算符优先级的设置,当从右向左扫描时遇到的第一个优先级最低且在括号外的运算符即为主运算符。

当这一切都实现了之后,表达式求值就可以正确地进行所有运算了。

```
(nemu) p (1+(3*2) +(($eax-$eax) +(*$eip-1**$eip)+(0x5--5+ *0X100005 - *0x100005) )*4)
Expression Value is dec:47, hex:0x0000002f
```

图 2-11 扩展后的求值示例

实现监视点

监视点实现较为简单，我们所做的就是利用一个链表来存放所有的监视。

首先我们有一个监视点池，`_free` 空闲监视点链表的头节点。当创建监视点时，从 `_free` 中取出第一个节点，将监视点内容作为 `string` 存放在其中，同时将当前表达式的值存放其中。每当执行完一条新的指令，就会在 `cpu_exec` 函数中调用 `hit_wp_check`，该函数会遍历监视点链表，对每一个监视点进行求值并与旧值进行比较，当发生变化时就命中了该监视点，将 `cpu` 状态设置为 `stop`，程序运行暂时停止。

当删除监视点时，只需遍历 `head` 链表，将对应的监视点从中取出放入到 `_free` 即可。

断点

断点的实现主要利用监视点即可，将监视的内容设置为 `$eip==addr`，这里的 `addr` 即为你要设置的断点地址。

```
(nemu) w $eip==0x100005
[src/monitor/debug/ui.c,104,cmd_w] Watchpoint id:0,now value:0x00000000
(nemu) c
[src/monitor/debug/watchpoint.c,65,hit_wp_check] Hit watchpoint 0:value from 0x00000000 to 0x00000000
1.
```

图 2-12 监视点

至此，我们的简易调试器便已全部完成。

2.4 必答题

理解基础设施

$$500 \times 0.9 \times 20 \times (30 - 10) / 3600 = 50h$$

按照假设，理论上可以节省 50 个小时的调试时间（事实上并没有）。

查阅 i386 手册

EFLAGS.CF：阅读范围 2.3.4Flags Register

ModR/M：阅读范围 17.2.1ModR/M and SIB Bytes

Mov：阅读范围 17.2.2.11 Instruction Set Detail

shell 命令

```
find . -name '*.h|c'|xargs wc -l
```

```
find . -name '*.h|c'|xargs cat|grep -v ^$|wc -l
```

*

使用 **man**

-Wall 打开所有警告

-Werror 将所有警告视作错误

3 PA2 冯诺依曼计算机

3.1 第一个 C 程序

在 `nexus-am/tests/cputest` 下，执行 `make ALL=dummy run`，即可使 `nemu` 加载我们的第一个 c 程序 `dummy`，该程序的源文件是 `nexus-am/tests/cputest/dummy/dummy.c`，在该程序中，`main` 函数什么都没做就直接返回。

因此，该程序包含了一个程序运行所需要指令的最小子集，我们只需实现少量的几条指令如 `ret`, `call`, `mov`, `push`, `pop`, `xor` 等即可。

每当遇到一条未实现的指令，首先要到 `i386` 手册的指令集中查阅该指令的操作数类型以及行为描述，之后在 `opcode_table` 中补充对应的译码函数和执行函数，最后完善执行函数，利用 `rtl` 指令进行寄存器和内存操作。

当实现了所有指令之后，我们再运行 `dummy` 时可以看到成功执行的信息 `HIT GOOD TRAP`。

```
void@ubuntu:~/ics2018/nexus-am/tests/cputest$ make ALL=dummy run
Building dummy [x86-nemu] with AM_HOME {/home/void/ics2018/nexus-am}
Building am [x86-nemu]
Building klib [x86-nemu]
Building compiler-rt [x86-nemu]
[src/monitor/monitor.c,73,load_img] The image is /home/void/ics2018/nexus-am/tests/cputest/build/dummy-x86-nemu.bin
[src/monitor/monitor.c,34,welcome] Debug: OFF
[src/monitor/monitor.c,37,welcome] Build time: 19:03:00, Jan 7 2019
Welcome to NEMU!
For help, type "help"
nemu: HIT GOOD TRAP at eip = 0x0010001b
[src/monitor/cpu-exec.c,24,monitor_statistic] total guest instructions = 13
dummy
```

图 3-1 dummy 运行结果

3.2 丰富指令集

在该环节中，我们所面对的不只是一个文件，而是 `cputest` 下的所有文件，我们可以按任意顺序逐个实现这些文件运行所需指令，其中 `string.c` 和 `hello-str.c` 因为涉及到 `string.h` 和 `stdio.h` 库函数的编写，可以暂时搁置。

需要注意的是，我们可以先实现 `diff-test`，再进行指令补充。由于 `guide-book` 上的顺序安排，我个人一直到所有文件通过之后才发现有 `diff-test` 这种 `debug` 神器，不仅浪费了大量的时间进行肉眼 `debug`，同时一些指令的错误实现并没有被揭露出来，为后续的实验埋下了许多坑。

`Diff-test` 的实现十分简单，只需要在 `common.h` 中声明 `DIFF_TEST` 宏，并在

nemu/tools/diff-test/diff-test.c 中补全 `diffest-step` 函数即可，我们可以借助 `diffest_getregs` 获取到 `nemu` 的运行状态 `dut_r`，再将其与通过 `ref_diffest_getregs` 获取到的 `qemu` 寄存器状态进行比较，若所有寄存器值相同则通过比较，否则将 `nemu` 状态设为 `NEMU_ABORT`。

非字符串操纵程序都实现后，我们可以去利用 `_putc` 实现 `stdio.h` 和 `string.h` 两个 `klib` 函数库，最终在 `nemu` 下执行 `shell` 脚本 `runall.sh`，可以看到所有文件的通过情况。

```
void@ubuntu:~/ics2018/nemu$ bash runall.sh
compiling NEMU...
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ arith-bit] PASS!
[ bubble-sort] PASS!
[ div] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shuixianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

图 3-2 一键回归测试

3.3 I/O 指令

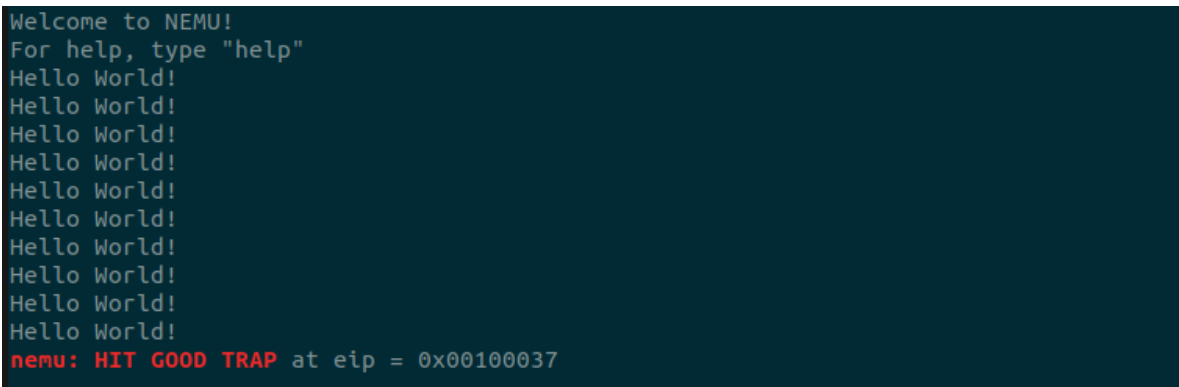
在这一步中，我们需要实现 io 相关指令 `in` 和 `out`，并实现时钟，键盘和 `vga` 三种 ioe。

In 和 out

`In` 和 `out` 两条指令本身很容易实现，只需要根据译码得到的信息去读写对应内存就可以了。

当成功实现之后，我们的 `_putc` 就能够将字符成功的输出到屏幕上，这也就意味着我们自己编写的 `klib/stdio` 库函数 `printf` 能够正常工作。

在 `nexus-am/apps/hello` 下 `make run`，可以看到输出。



```
Welcome to NEMU!  
For help, type "help"  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
nemu: HIT GOOD TRAP at eip = 0x00100037
```

图 3-3hello 运行结果

IOE

该环节需要分别实现时钟，`keyboard` 和 `vga`，这三者的实现都十分简单，前两者只需借助 `inl` 从对应的端口 `0x48` 和 `0x60` 读出时间或者键盘输入，在加以处理即可。而 `vga` 中则可以直接对 `fb` 数组进行操作，将对应坐标的像素矩阵正确写入。这些都实现正确后，可以进行对应的测试。

跑分

当实现了时钟后，我们就可以通过 `microbench` 来测试 `nemu` 的性能。

*

```
Welcome to NEMU!
For help, type "help"
[qsort] Quick sort: * Passed.
      min time: 790 ms [698]
[queen] Queen placement: * Passed.
      min time: 956 ms [539]
[bf] Brainf**k interpreter: * Passed.
      min time: 6055 ms [432]
[fib] Fibonacci number: * Passed.
      min time: 10747 ms [265]
[sieve] Eratosthenes sieve: * Passed.
      min time: 10451 ms [405]
[15pz] A* 15-puzzle search: * Passed.
      min time: 2099 ms [275]
[dinic] Dinic's maxflow algorithm: * Passed.
      min time: 1754 ms [771]
[lzip] Lzip compression: * Passed.
      min time: 5251 ms [504]
[ssort] Suffix sort: * Passed.
      min time: 1010 ms [585]
[md5] MD5 digest: * Passed.
      min time: 10379 ms [188]
=====
MicroBench PASS          466 Marks
                        vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 3-4microbench 跑分

红白机

当所有环节都正确实现，就可以尝试运行超级玛丽。



图 3-5 超级玛丽运行图

3.4 必答题

编译与链接 在 `nemu/include/cpu/rtl.h` 中, 你会看到由 `static inline` 开头定义的各种 RTL 指令函数. 选择其中一个函数, 分别尝试去掉 `static`, 去掉 `inline` 或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

单独去掉 `inline` 会报错, 去掉 `static` 不会报错, 同时删去二者会报错。

当单独删去 `inline` 时, 因为 `prefic.c` 包含了该函数却没有使用, 因此报错。

当同时删去 `inline` 和 `static` 时, 会出现函数重定义, 因此报错。

编译与链接

1. 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy;` 然后重新编译 NEMU. 请问重新编译后的 NEMU 含有多少个 `dummy` 变量的实体? 你是如何得到这个结果的?

通过 `grep` 可以计算出 `common.h` 在 `nemu/build/obj` 路径下一共出现 77 次, 加上 `common.h` 本身, 一共 78 个实体。

2. 添加上题中的代码后, 再在 `nemu/include/debug.h` 中添加一行 `volatile static int dummy;` 然后重新编译 NEMU. 请问此时的 NEMU 含有多少个 `dummy` 变量的实体? 与上题中 `dummy` 变量实体数目进行比较, 并解释本题的结果.

30 个, 计算过程同上。

3. 修改添加的代码, 为两处 `dummy` 变量进行初始化: `volatile static int dummy = 0;` 然后重新编译 NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)

会出现重定义的错误。

4 PA3 批处理系统

4.1 中断与上下文

4.1.1 中断机制

我们首先要实现 i386 中断机制，也就是加入对 `int` 和 `lidt` 指令的支持，其中 `lidt` 指令是将中断表的入口地址放入寄存器 `idtr`，而 `int` 指令则是根据将所有寄存器状态保存在栈中，然后根据具体的中断号在 `idtr` 指向的中断跳转表中寻找到对应的中断处理程序，然后进行跳转。

为了方便后续的使用，我们将中断机制的执行放在函数 `raise_intr` 中进行，而 `int` 指令本身只负责调用该函数。

4.1.2 保存上下文

首先我们要实现 `pusha` 指令，该指令负责在中断处理程序中进行一些必要的压栈。

其次，我们需要根据各个寄存器以及中断号等数据压栈的顺序组织成一个结构 `_Context`，该结构中的数据所在的内存实质上就是触发中断并压栈后栈中保存上下文的数据所对应的内存。

当我们以正确的顺序实现该结构体，就可以在 `irq_handle` 中正确的获取到中断号，以进行事件处理。

4.1.3 事件分发

这一步很简单，只需要在 `irq_handle` 中针对传入的 `irq` 进行判断，当为 `0x80` 时将事件封装为 `SYSCALL`，`0x81` 时封装为 `YIELD`，之后交由 `do_event` 进行处理。

在 `do_event` 中，暂时只需要处理 `YIELD` 事件。

4.1.4 恢复上下文

执行中断处理程序成功之后，我们需要能够返回到中断之星前的状态继续运行程序，所以需要实现 `popa` 和 `iret` 两条指令。

当这些都完成之后，`dummy` 程序就可以正确执行了。

```
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 20:16:55, Jan  7 2019
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1056352 =
[src/device.c,49,init_device] Initializing devices...
[src/irq.c,26,init_irq] Initializing interrupt/exception handler
Sys_yield.own
Event_yield.
Pageup
```

图 4-1dummy 触发的 yield

4.2 系统调用

4.2.1 实现 loader

Loader 函数将根据传入的 filename 调用 fs_open 来加载具体的文件，不过在这个阶段，我们还未实现文件系统，因此只需要将这些参数忽略，直接将整个 ramdisk 加载到以 0x4000000 地址起始的内存中。

Loader 的实现可以借助 ramdisk_read，在实现文件系统后也可借助与 fs_read 进行内存加载。

当实现了 loader 后，需要在 init_proc 中调用 naïve_unload，该函数的定义在 loader.c 中，它会调用 loader 加载程序到 0x4000000，之后从该地址开始执行程序。

4.2.2 识别系统调用

在这个阶段，我们需要能够在 irq_handle 中根据 irq 中断号判断，当为 0x80 将事件封装为 EVENT_SYSCALL 并交由 do_event 处理。

在 do_event 中，我们会对 SYSCALL 的事件调用 do_syscall，并将上下文 Context 作为参数传入。

Do_syscall 的函数定义在 syscall.c 中，我们需要在这个函数中读出上下文中的四个寄存器的值，其中第一个寄存器也就是 eax 的值表示了本次系统调用的类型，我们需要根据其进行判断，分别进行不同的操作。

4.2.3 基本系统调用

SYS_yield: 该调用直接执行 _yield() 即可。

SYS_exit: 该系统调用直接执行 `_halt()` 即可，在实现了文件系统和批处理系统后将会改为调用 `naïve_upload` 加载“bin/init”程序。

SYS_write: 该系统调用直接调用将 `a[2]` 指向的数据写入 `a[1]`，长度为 `a[3]` 个字节，在当前阶段直接调用 `_putc` 输出即可。

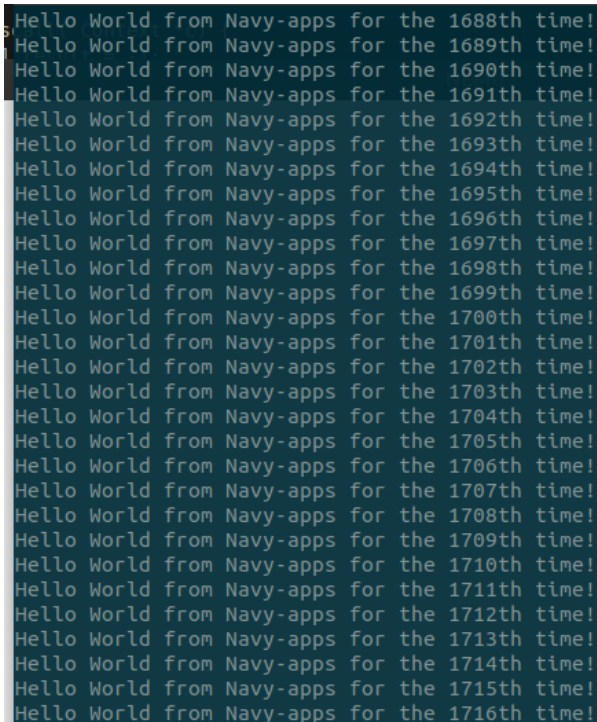
当实现了 `write` 系统调用，运行 `/bin/hello` 时就能够成功的输出字符串在屏幕上，不过因为此时还没有实现堆区管理，无法进行内存分配，所以每次调用只能传入一个字符，也就是说每次 `write` 系统调用只会发生一次 `_putc`。

4.2.4 堆区管理

通过 `man`，我们可以了解到 `brk` 和 `sbrk` 的详细信息，之后需要借助于 `_end` 来对堆区进行管理。

每次执行 `SYS_brk` 系统调用时，都会获得参数 `increment` 也就是堆区新分配的字节数，我们需要将 `_end` 也就是 `program break` 增加这么多字节，以此标识堆区内存的分配与释放。

实现完成后，我们就能够正确的进行堆内存分配与释放了，此时再执行 `/bin/hello` 时，就能看到每次进行 `write` 调用可以成功输出一整句话而不再是之前的一个字符。



```
Hello World from Navy-apps for the 1688th time!
Hello World from Navy-apps for the 1689th time!
Hello World from Navy-apps for the 1690th time!
Hello World from Navy-apps for the 1691th time!
Hello World from Navy-apps for the 1692th time!
Hello World from Navy-apps for the 1693th time!
Hello World from Navy-apps for the 1694th time!
Hello World from Navy-apps for the 1695th time!
Hello World from Navy-apps for the 1696th time!
Hello World from Navy-apps for the 1697th time!
Hello World from Navy-apps for the 1698th time!
Hello World from Navy-apps for the 1699th time!
Hello World from Navy-apps for the 1700th time!
Hello World from Navy-apps for the 1701th time!
Hello World from Navy-apps for the 1702th time!
Hello World from Navy-apps for the 1703th time!
Hello World from Navy-apps for the 1704th time!
Hello World from Navy-apps for the 1705th time!
Hello World from Navy-apps for the 1706th time!
Hello World from Navy-apps for the 1707th time!
Hello World from Navy-apps for the 1708th time!
Hello World from Navy-apps for the 1709th time!
Hello World from Navy-apps for the 1710th time!
Hello World from Navy-apps for the 1711th time!
Hello World from Navy-apps for the 1712th time!
Hello World from Navy-apps for the 1713th time!
Hello World from Navy-apps for the 1714th time!
Hello World from Navy-apps for the 1715th time!
Hello World from Navy-apps for the 1716th time!
```

图 4-2write 系统调用

4.3 文件系统

4.3.1 简易文件系统

在此阶段，我们所说的文件系统还只服务于通常意义上的文件。

首先我们要对 `Finfo` 结构进行一定的修改，加入成员变量 `open_offset`，当文件未被打开时，该变量的值是 0，当文件被打开之后，该变量将会指示文件读写指针当前的偏移值。

我们需要实现 `fs_open`，`fs_read`，`fs_close` 等函数。

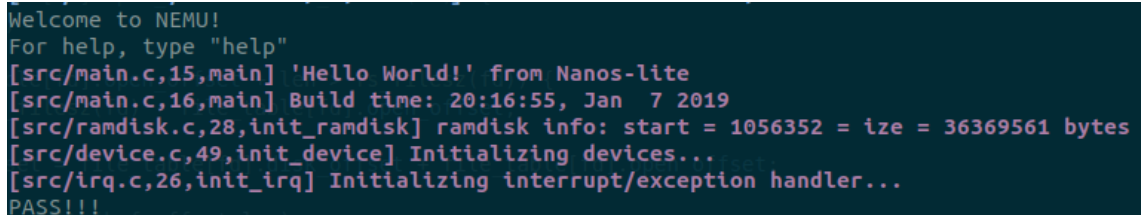
`fs_open` 中，我们可以对 `file_table` 进行遍历，当找到对应 `filename` 的文件时将其索引作为 `fd` 返回，否则返回 -1。

`fs_read` 中，我们需要根据传入的 `fd` 和 `len` 来对文件进行读入，即将内存地址 `file_table[fd].disk_offset + file_table[fd].open_offset` 为起始的 `len` 个字节读入 `buf`。

因为我们只是模拟了文件的打开关闭，因此 `fs_close` 直接返回 0 即可。

其它一些文件操作函数的执行过程大体同上，只需要把握好 `open_offset` 的含义即可顺利写出。

当全部内容完成之后，我们可以尝试运行 `/bin/text`，如一切正常，则会输出 `pass`。



```
Welcome to NEMU!  
For help, type "help"  
[src/main.c,15,main] 'Hello World!' from Nanos-lite  
[src/main.c,16,main] Build time: 20:16:55, Jan 7 2019  
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1056352 = ize = 36369561 bytes  
[src/device.c,49,init_device] Initializing devices...  
[src/irq.c,26,init_irq] Initializing interrupt/exception handler...  
PASS!!!
```

图 4-3text 测试

4.3.2 虚拟文件系统

虚拟文件系统的核心思想便是“一切皆文件”。

在虚拟文件系统中，除了通常意义上的文件之外，我们也会将各种 `ioc` 视作文件，统一由 `fs_open`，`fs_read` 系列函数进行操作。

能够做到这一点主要在与 `Finfo` 的读写函数指针，对于不同的设备有不同的读写函数，我们将其统一封装传入 `Finfo` 中的函数指针，再由 `fs` 操作函数同意调用，即可实现文件或者设备在接口调用上的统一。

在完成 `vga` 之后，我们可以执行 `/bin/bmptest`，实现正确的情况下将会输出 `nanolite`

的 logo。



图 4-4nemu logo

把设备输入抽象成文件之后，则可执行/bin/events，能够获取键盘事件。

```
receive event: t 6208
receive event: t 6371
receive event: t 6535
receive event: t 6697
receive event: t 6873
receive event: t 7035
receive event: kd W
receive event: ku W
receive event: t 7528
receive event: kd S
receive event: ku S
receive event: t 8051
receive event: t 8227
receive event: t 8420
receive event: t 8606
receive event: t 8794
```

图 4-5event 按键响应

4.4 批处理系统

在这最后的环节中，我们要做的工作不多。

如果一切实现正确，仙剑奇侠传就可以正常运行了。

此外，我们通过对 `exit` 系统调用和 `proc_init` 进行一点小小的修改，并实现 `execve` 系统调用，可以使得程序运行前和运行结束都返回到 `/bion/init` 继续执行而非直接退出。

*

这样，也就实现了一个简单的批处理系统。

展示如下。



图 4-6 批处理系统界面



图 4-7 仙剑主界面



图 4-8 仙剑人物对话



图 4-9 仙剑内游戏菜单

4.5 必答题

文件读写的具体过程 仙剑奇侠传中有以下行为：

在 navy-apps/apps/pal/src/global/global.c 的 PAL_LoadGame() 中通过 fread() 读取游戏存档

在 navy-apps/apps/pal/src/hal/hal.c 的 redraw() 中通过 NDL_DrawRect() 更新屏幕
请结合代码解释仙剑奇侠传，库函数，libos, Nanos-lite, AM, NEMU 是如何相互协助，来分别完成游戏存档的读取和屏幕的更新。

首先，屏幕的更新是通过 `NDL_DrawRect()` 函数来进行的。

`NDL_DrawRect()` 定义于 `navy-apps/libs/libndl/src/ndl.c`，它通过调用 `fwrite()` 来进行写入。而 `fwrite` 是一个 C 语言库函数，在 `navy-apps/libs/libc/src/stdio/fwrite.c` 实现，通过 `fwrite()-> __sfvwrite -> _write()` 这样的途径，最终调用了 `navy-apps/libs/libos/src/nanos.c` 中的 `_write` 接口，从而调用 `system.c` 中的 `SYS_write`。

随后，我们在 `do_syscall` 中判断出 `SYS_write` 从而调用 `fs_write()`，在该函数中判断出 `fd` 类型为 `FD_FB`，于是通过文件读写函数指针调用了 `fb_wite`，在该函数中通过 `draw_rect()` 来进行 VGA 绘制。

总的来看，应用程序的整体流程大致如下：在 `nanos-lite` 执行 `make run`，首先程序编译 `navy-apps` 路径下我们指定的应用程序，而在编译应用程序时使用的系统调用有关的实现均在 `navy-apps/libs/` 下代码中实现。另一部分代码会通过 `nexus-am/am/arch/x86-nemu/src/` 中定义的硬件相关的代码来实现对硬件的具体读写等操作，最终在应用程序、操作系统、`lib`、`am` 等完成编译后生成镜像，转由 `nemu` 执行。

5 设计总结与心得

5.1 课设总结

本次课设的难度可以说是三年来最高的一次，收获也是最大的一次。

从 PA0，到 PA3，光是写代码和调试便花费了差不多五十个小时，更不用说阅读框架代码和 i386 手册的时间。

在 PA1 中，通过实现简易调试器，涉及到了 gdb 的使用，正则表达式，链表，string 库函数等知识，可以说是对于 c 语言和数据结构的一次回顾与总结。

在 PA2 中，侧重点则主要在于 x86 指令的实现，涉及到了组成原理和汇编的许多知识。

PA3 中，通过实现系统调用，文件系统，可以说是对与操作系统的知识有了一次全面的回顾。

通过本次课设，对于过去三年的专业课知识，我获得了不少新的认识，一些已经遗忘的知识被重新捡了起来，同时也对于一个完善的大型项目如何入手，如何阅读有了新的理解。

5.2 课设心得

本次课设的心得主要在于两点。

一是要全局考虑，在 PA3 的开发过程中往往会遇到一些不清楚实现细节的任务，这些往往在当前文件下无法找到实现方法，这时候就需要仔细通读整套框架的代码，有许多细节才能被注意到。

二是重视基础设施建设，例如 PA1 中简易调试器的搭建就为后续的调试省了许多功夫，尤其是监视点的存在，使得能够较容易地追踪和复现问题。其次就是 diff-test，因为 guidebook 中 difftest 的顺序是在实现所有指令之后，因此在实现指令的过程中 debug 花费了几乎三十个小时的时间，但是一旦使用了 difftest，就能大幅度缩减追踪问题所花费的时间。

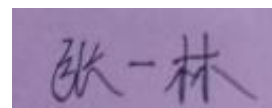
• 指导教师评定意见 •

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：



二、对课程设计的学术评语（教师填写）

三、对课程设计的评分（教师填写）

评分项目 (分值)	报告撰写 (30 分)	课设过程 (70 分)	最终评定 (100 分)
得分			

指导教师签字：_____