

# 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: CS1506

学 号: U201514639

姓 名: 刘科翰

电 话: 13018052550

邮 件: 524792973@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

<b>1</b>	<b>课程设计概述.....</b>	<b>2</b>
1.1	设计任务 .....	2
<b>2</b>	<b>PA1 简易调试器.....</b>	<b>3</b>
2.1	功能实现的要点.....	3
2.2	必答题 .....	6
2.3	主要故障与调试.....	8
<b>3</b>	<b>PA2 冯诺依曼计算机系统.....</b>	<b>10</b>
3.1	功能实现的要点.....	10
3.2	必答题 .....	12
3.3	主要故障与调试.....	15
<b>4</b>	<b>PA3 批处理系统.....</b>	<b>16</b>
4.1	功能实现的要点.....	16
4.2	必答题 .....	19
4.3	主要故障与调试.....	19
<b>5</b>	<b>实验总结与心得 .....</b>	<b>21</b>

## 1 课程设计概述

### 1.1 设计任务

Programming Assignment:

PA0: 世界诞生的前夜: 开发环境的配置。

PA1: 开天辟地的篇章: 最简单的计算机。

task PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存

task PA1.2: 实现算术表达式求值

task PA1.3: 实现所有要求, 提交完整的实验报告

PA2: 简单复杂的机器: 冯诺依曼计算机系统。

task PA2.1: 在 NEMU 中运行第一个 C 程序 dummy

task PA2.2: 实现更多的指令, 在 NEMU 中运行所有 cputest

task PA2.3: 运行打字小游戏, 提交完整的实验报告

PA3: 穿越时空的旅程: 批处理系统。

task PA3.1: 实现自陷操作 `_yield()` 及其过程

task PA3.2: 实现用户程序的加载和系统调用, 支撑 TRM 程序的运行

task PA3.3: 运行仙剑奇侠传并展示批处理系统, 提交完整的实验报告

PA4: 虚实交错的魔法: 分时多任务。

task PA4.1: 实现基本的多道程序系统

task PA4.2: 实现支持虚存管理的多道程序系统

task PA4.3: 实现抢占式分时多任务系统, 并提交完整的实验报告

PA5: 天下武功唯快不破: 程序与性能。

## 2 PA1 简易调试器

### 2.1 功能实现的要点

#### 2.1.1 task PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存等基本指令

如表 2-1 所示, 这是本次简易调试器需要实现的基本命令。

命令	格式	说明
帮助(1)	help	打印命令的帮助信息
继续运行(1)	c	继续运行被暂停的程序
退出(1)	q	退出 NEMU
单步执行	si [N]	让程序单步执行 N 条指令后暂停执行, 当 N 没有给出时, 缺省为 1
打印程序状态	info SUBCMD	打印寄存器状态 打印监视点信息
表达式求值	p EXPR	求出表达式 EXPR 的值, EXPR 支持的 运算请见调试中的表达式求值小节
扫描内存(2)	x N EXPR	求出表达式 EXPR 的值, 将结果作为起始内存 地址, 以十六进制形式输出连续的 N 个 4 字节
设置监视点	w EXPR	当表达式 EXPR 的值发生变化时, 暂停程序执行
删除监视点	d N	删除序号为 N 的监视点

表 2-1 简易调试器需要实现的基本命令

对于有操作数的指令, 需要对指令进行解析, 具体使用到的函数是 strtok 函数将参数提取出来, 并且进行指令格式的判断。

单步调试需要了解程序的运行的逻辑, 可以参考 c 指令的实现, 需要调用 cpu\_exec 函数, 并传入执行的指令数, 里面有一个 for 循环, 调用对应次数的 exec\_wrapper 函

# 华中科技大学课程设计报告

---

数，执行对应数目的指令。至于执行中的信息在函数中已有相关的输出。

打印寄存器的内容，直接读取并输出 `cpu` 结构体变量中的各个部分的内容即可。

扫描内存使用了 `paddr_read` 函数，解析出对应的地址，调用函数读取并输出即可。

## 2.1.2 task PA1.2: 实现简单的算术表达式求值

为了实现简单的表达式求值，需要先书写一般情况需要识别的正则表达式，在正则表达式数组中，如十进制数字、左右括号、空格、加减乘除运算符、逻辑运算符等等。由于在识别 `token` 时，遍历数组，先匹配先得，故数组内的匹配规则越靠前证明这个规则代表的元素的优先级越高。

遍历完得到 `tokens` 数组后，为了区分解引用和符号这两种符号，应该再次遍历 `tokens` 数组，将符合单目运算符特点的乘号和减号的 `token` 类型改为解引用类型 `DEREF` 和负号类型 `NEGATIVE`。

为了实现简单的算术表达式求值，需要有对表达式进行格式解析的函数，如检查表达式是否合法，有没有括号过多的情况；如检查是否满足表达式 BNF 范式，BNF 范式的特定是，两边的 `token` 除了空格外必须是一对匹配的括号，内部的表达式的括号必须也是匹配合法的，但不需要满足 BNF 范式。另外，在格式解析中，最重要的是排除空格这类型的影响，因此要编写一个除去空格的函数。

之后在表达式求值函数中要分类讨论，进行递归求值，分为不合法的情况、满足 BNF 范式的情况、只有单个 `token` 可以直接取值的情况、表达式过长需要找主运算符拆分递归求值后归并的情况。

对于表达式过长需要找主运算符拆分递归求值后归并的情况，重要的是寻找主运算符函数的编写，在函数实现过程中除了要满足主运算符的特点外还要十分注意运算符之间的优先级。

## 2.1.3 task PA1.3: 实现扩展的表达式求值和监视点功能，完善指令系统

为了对表达式求值进行扩展，通过正则表达式加入了对十六进制数、寄存器、逻辑运算符 `!=`、`&&` 的识别，然后进一步完善表达式求值函数，从而实现扩展的表达式求值。进而可以实现 `p` 指令，`p` 指令通过传入表达式，调用我们写好的表达式求值函数即可返回结果。

对于正则表达式，在此列出所有用到的正则表达式（使用 `\` 是为了在字符串中实

# 华中科技大学课程设计报告

现正确的转义，对于解引用和符号的 token 类型在后面进行解析并修改)：

```
{ " +", TK_NOTYPE }, // 空格
{ "\"\\(", TK_LP }, // 左括号
{ "\"\\)", TK_RP }, // 右括号
{ "0x[0-9a-fA-F]+|0X[0-9a-fA-F]+u?", TK_HNUM }, // 16 进制数
{ "[0-9]+u?", TK_ONUM }, // 10 进制数
{ "\\$[a-z]+", TK_REG }, // 没有限制字母数字，不做判断，由软件判断
{ "\\\\", '/' }, // divide '/'
{ "\\*", '*' }, // multiply '*'
{ "\\-", '-' }, // minus '-'
{ "\\+", '+' }, // plus '+'
{ "==", TK_EQ }, // equal '='
{ "!=", TK_NEQ }, // noequal '!='
{ "&&", TK_AND } // and, '&&'
```

关于监视点，就是进行链表的管理，info 指令可以在链表中读取监视点信息。每条指令执行完毕后在 cpu\_exec 函数中检查其变化，如果产生变化，将 nemu 状态设为 NEMU\_STOP，然后 return 出 cpu\_exec 函数，接下来通过完善后 info 指令，查看当前监视点的状态变化。如图 2-1、图 2-2、图 2-3 所示，这是本次扩展后的表达式求值的测试结果节选。

```
For help, type "help"
(nemu)
(nemu) p (1+(3*2))+(($eax-$eax)+(*$eip-1**$eip)+(0x5--5+ *0X100005 - *0x100005) ) *4)
[src/monitor/debug/expr.c,95,make_token] match rules[1] = "\"\\(" at position 0 with len 1: (
[src/monitor/debug/expr.c,95,make_token] match rules[4] = "[0-9]+u?" at position 1 with len 1: 1
[src/monitor/debug/expr.c,95,make_token] match rules[9] = "\\+" at position 2 with len 1: +
[src/monitor/debug/expr.c,95,make_token] match rules[1] = "\"\\(" at position 3 with len 1: (
```

图 2-1 复杂表达式的求值

```
[src/monitor/debug/expr.c,95,make_token] match rules[2] = ")" at position 73 with len 1: )
[src/monitor/debug/expr.c,95,make_token] match rules[7] = "\\*" at position 74 with len 1: *
[src/monitor/debug/expr.c,95,make_token] match rules[4] = "[0-9]+u?" at position 75 with len 1: 4
[src/monitor/debug/expr.c,95,make_token] match rules[2] = "\\)" at position 76 with len 1: )

result: 47 (hex value:0x0000002f)
(nemu) □
```

图 2-2 图 2-1 中表达式求值结果

# 华中科技大学课程设计报告

```
(nemu) w $eip == 0x100005
[src/monitor/debug/expr.c,95,make_token] match rules[5] = "$[a-z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,95,make_token] match rules[0] = "+" at position 4 with len 1:
[src/monitor/debug/expr.c,95,make_token] match rules[10] = "==" at position 5 with len 2: ==
[src/monitor/debug/expr.c,95,make_token] match rules[0] = "+" at position 7 with len 1:
[src/monitor/debug/expr.c,95,make_token] match rules[3] = "0x[0-9a-fA-F]+|0X[0-9a-fA-F]+u?" at position 8 with len 8: 0x100005

WatchPoint 0 has set successfully!
(nemu) w $eax
[src/monitor/debug/expr.c,95,make_token] match rules[5] = "$[a-z]+" at position 0 with len 4: $eax

WatchPoint 1 has set successfully!
(nemu) w $ecx
[src/monitor/debug/expr.c,95,make_token] match rules[5] = "$[a-z]+" at position 0 with len 4: $ecx

WatchPoint 2 has set successfully!
(nemu) info w
Num  Expr                               Oldvalue  Value
0    $eip == 0x100005                   0x0       0x0
1    $eax                               0x0       0x5ede4d05
2    $ecx                               0x0       0x1de06507
(nemu) si 2
100000:  b8 34 12 00 00                      movl $0x1234,%eax
[src/monitor/debug/expr.c,95,make_token] match rules[5] = "$[a-z]+" at position 0 with len 4: $eip
[src/monitor/debug/expr.c,95,make_token] match rules[0] = "+" at position 4 with len 1:
[src/monitor/debug/expr.c,95,make_token] match rules[10] = "==" at position 5 with len 2: ==
[src/monitor/debug/expr.c,95,make_token] match rules[0] = "+" at position 7 with len 1:
[src/monitor/debug/expr.c,95,make_token] match rules[3] = "0x[0-9a-fA-F]+|0X[0-9a-fA-F]+u?" at position 8 with len 8: 0x100005

WatchPoint 0 has changed!
[src/monitor/debug/expr.c,95,make_token] match rules[5] = "$[a-z]+" at position 0 with len 4: $eax

WatchPoint 1 has changed!
[src/monitor/debug/expr.c,95,make_token] match rules[5] = "$[a-z]+" at position 0 with len 4: $ecx

(nemu) □
```

图 2-3 w 指令、info w 指令以及 si 指令的因监视点改变中断执行

## 2.2 必答题

1.理解基础设施。我们通过一些简单的计算来体会简易调试器的作用。首先作以下假设：假设你需要编译 500 次 NEMU 才能完成 PA。假设这 500 次编译当中，有 90% 的次数是用于调试。假设你没有实现简易调试器，只能通过 GDB 对运行在 NEMU 上的客户程序进行调试。在每一次调试中，由于 GDB 不能直接观测客户程序，你需要花费 30 秒的时间来从 GDB 中获取并分析一个信息。假设你需要获取并分析 20 个信息才能排除一个 bug。那么这个学期下来，你将会在调试上花费多少时间？由于简易调试器可以直接观测客户程序，假设通过简易调试器只需要花费 10 秒的时间 从中获取并分析相同的信息。那么这个学期下来，简易调试器可以帮助你节省多少调试的时间？

答：（1）没有简易调试器下的情况：

一共要进行 450 次调试，600s 排除一个 bug，假设一次调试解决一个 bug，那么在完成 nemu 的过程中，需要花费 4500 分钟来调试，也即 75 个小时用于调试。

（2）有简易调试器下的情况：

解决一个 bug 需要 200s，则完成 nemu 过程中需要 25 个小时用于调试。

# 华中科技大学课程设计报告

2. 查阅 i386 手册。

(1) EFLAGS 寄存器中的 CF 位是什么意思？

(2) ModR/M 字节是什么？

(3) mov 指令的具体格式是怎么样的？

答：(1) 阅读第二章 Basic Programming Model 中的 2.3 章节 Register 其中的 2.3.4 小节 Flag Register，在 2.3.4.1 中有一个给 Appendix C 的跳转链接，里面详细介绍了每一个标志物的定义。CF，全称 Carry Flag，是一个 Status Flags——状态标志位，状态标志位使得当前指令的执行能够受到之前指令的结果的影响。CF 是标志高位进位或是借位信息的一个标志位。

(2) i386 手册 17.2.1 ModR/M and SIB Bytes 小节中介绍了 ModR/M and SIB 字节的含义和作用，其中 ModR/M 字节一般紧跟着指令操作码，这个字节一般包含着三个方面的信息，一共分为三个区域：分别为 mod 区、reg 区，r/m 区。mod 区占 3 个 bits，与 r/m 区的信息结合起来可以指示 8 个寄存器和 24 个索引模式，一个 32 种可能的情况。reg 区占 3 个 bits，可以用来表示一个寄存器也可以作为操作码 opcode 的补充，而这个由这条指令的第一个字节 opcode 决定。而 r/m 区占最后的 3 个 bits，可以指向一个作为操作数的寄存器，或者可以成为寻址模式中的一部分编码与 mod 区一起发挥作用。具体的字节信息如图 2-4 所示：

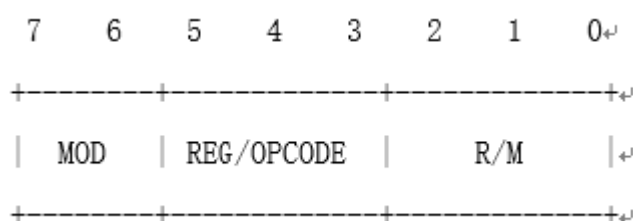


图 2-4 Modrm 字节结构

(3) i386 手册 17.2.2.11 Instruction Set Detials 里面详细地讲解了每一种指令，具体到 MOV 中，MOV 指令的格式为：MOV DES, SRC，MOV 指令将第二个操作数 SRC 的值拷贝到第一个操作数 DES 处。

3. shell 命令。

(1) 完成 PA1 的内容之后，nemu/ 目录下的所有.c 和.h 文件总共有多少行代码？

(2) 你是使用什么命令得到这个结果的？



# 华中科技大学课程设计报告

---

(3) 和框架代码相比, 你在 PA1 中编写了多少行代码?

(4) 除去空行之外, nemu/ 目录下的所有 .c 和 .h 文件总共有多少行代码?

答: (1) 完成 PA1 后, nemu/目录下总共有 4534 行代码 (包括空行、注释)。

(2) 使用的命令: `find . -name "*.ch" | xargs cat | wc -l`

(3) checkout 到 pa0 之后执行 (2) 中的命令, 得到的行数为 3833 行, 故在 pa1 中总共编写了 701 行代码。

(4) 除去空行之后为 3804/3114 行代码。使用的命令为: `find . -name "*.ch" | xargs cat | grep -v "^$" | wc -l`。可见编写代码中又多加了几行无用的空行。

4. 请解释 gcc 中的 -Wall 和 -Werror 有什么作用? 为什么要使用 -Wall 和 -Werror?

答: (1) -Wall 选项是一系列警告编译选项的集合, 本次实验中比较常见的警告有[-Wunused], [-Wunused]是也是一系列选项的集合, 用来警告存在一个定义了却未使用的局部变量或者一个函数的参数在函数的实现中并未被用到或者一个显式计算表达式的结果未被使用等等; 还有[-Wimplicit], 它也是一个选项的集合, 用于警告在声明函数却未指明函数返回类型时给出警告或者是在函数声明前调用该函数时给出警告等等。在-Wall 选项集合中还有很多其他的选项集合, 在编译调试时是很有用的工具。

(2) -Werror 是将所有的 Warning 都当成 Error 来处理, 视警告为错误, 只要有警告存在程序都会终止执行, 避免潜藏的 Fault。

## 2.3 主要故障与调试

### 2.3.1 寻找主运算符出错

出错原因: 忽略了运算符之间的优先级。

解决方案: 在编写寻找主运算符的函数的过程中, 除了排除括号内的运算符和非运算符, 剩下的就需要根据优先级进行选择, 具体优先级参考 c 语言各个运算符的优先级。

### 2.3.2 解引用在简单表达式可以用, 复杂则错误

出错原因: 未将解引用当成一个最高优先级的运算符来看待, 而是把它放在 eval 函数中的一个 if 分支中进行分流, 实现思想有误。

# 华中科技大学课程设计报告

---

解决方案：把解引用当成一个具有最高优先级的运算符，在寻找主运算符的过程中将它纳入。在进行计算时，由于它是单目运算符，需要与双目运算符进行区分。同样对于负号，也是这样处理的，先把它当成减号识别出来，然后在遍历 `tokens` 的时候将其区分为单目运算符，最后在寻找主运算符的函数中按照同样的方法解析出来即可。

## 2.3.3 计算表达式偶尔出错

出错原因：没有清空 `tokens` 数组中每一个元素的缓冲区，以至于还保留着上一次分析的 `token` 的信息。没有对有效 `token` 之间的空格 `token` 进行处理。

解决方案：每次 `make_tokens` 的时候，注意清空缓冲区。时刻注意空格的处理方式。

## 3 PA2 冯诺依曼计算机系统

### 3.1 功能实现的要点

#### 3.1.1 task PA2.1: 在 NEMU 中运行第一个 C 程序 dummy

我们需要填充一个类型为 `opcode_entry` 的结构体，以 `opcode` 作为下标，索引这个 `opcode table`，获取对应指令的 `opcode_entry`，这个结构体中包含了译码函数的函数指针、执行函数的函数指针、以及对应指令的操作数宽度。获取该结构体后，调用对应的译码函数、执行函数，并用操作数宽度对执行过程进行限制，从而实现指令的执行。

本次代码多次使用了宏，在填写 `opcode table` 的时候，利用了宏简化了、规范化了填写，一共采用了 5 个宏，分别为 `IDEXW`、`IDEX`、`EXW`、`EX`、`EMPTY`，默认填写 `EMPTY`，执行了一个 `exec_inv` 函数，这个 `exec_inv` 函数会自动终止 `nemu` 的运行，并且输出提示信息；对于 `IDEXW` 宏，是需要显式设置操作数宽度为 1 字节的时候才需要使用的，对于操作数宽度是 2 个或者 4 个字节的时候，使用的是 `IDEX` 宏，设置 `width` 为 0，然后在函数 `set_width` 根据指令是否有 `0x66` 前缀在 `decoding` 结构体变量设置操作数宽度标志位 `is_operand_size_16`。这里代码框架已经在 `opcode` 为 `0x66` 处设置对应的处理函数 `exec_operand_size`，帮助我们区分开操作数的宽度了，因此在实现的时候就不需要考虑如何解析前缀了。`EXW` 和 `EX` 对比之前两个宏就是没有调用译码函数的情况而已，对于某些指令或者某些二次填表的指令，是不需要译码的，因此定义了这两个宏。

`make_DHelper` 函数是 `decode` 译码函数的一个宏，通过查阅 x86 手册，得知指令的源操作数类型和目的操作数类型，从而确定使用哪个译码函数，不同的译码函数会取出指令中的数据，放在 `id_dest`、`id_src`、`id_src1` 这三个宏的某几个，这三个宏就代表了解码信息结构体变量中的三个操作数，只是简写了一下而已。本次代码框架提供了全部的 `decode` 函数，基本没有必要自己写 `decode` 函数。

`make_EHelper` 函数是 `exec` 执行函数的一个宏，通过查阅 x86 手册，得知指令执行的过程，如何影响 `eflags` 寄存器，从而实现指令的执行。在执行函数中使用 `rtl` 指令的组合实现具体的功能，因此要先实现各个基本的 `rtl` 指令。

# 华中科技大学课程设计报告

## 3.1.2 task PA2.2: 实现更多的指令, 在 NEMU 中运行所有 cputest 通过

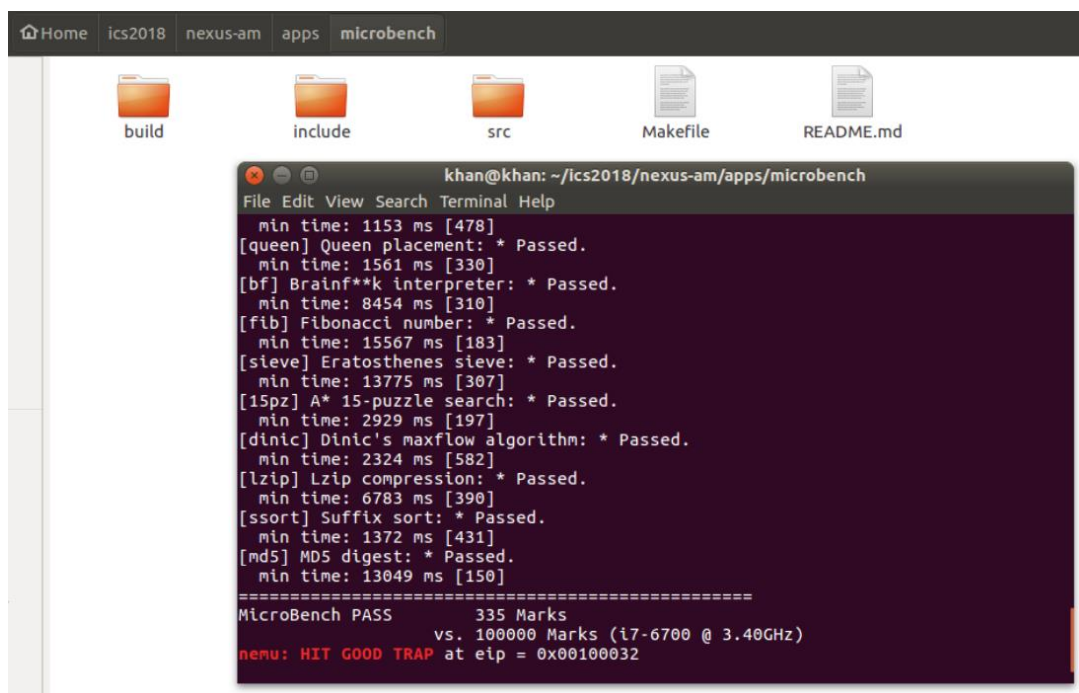
实现 Ddiffest, 现在 nemu/tools/qemu 处 make 出 qemu-so 的镜像, 才能使用 ddiffest, ddiffest 通过对比 DUT 和 REF 之间寄存器的值, 一旦出现不同, 立刻终止程序的运行, 这样就可以快速定位错误的具体位置, 从而高效地调试程序。

实现一些库函数。如 strlen、strcpy、strncpy、strcmp、strcat、strncmp、memcpy、memcmp、memset, 进而实现 sprintf、printf、snprintf, sprintf、printf、snprintf, 这三个函数可以调用一个通用的 vsprintf 函数, 通过输入不同的参数实现不同的效果, 一定程度实现解耦。vsprintf 需要实现关于 %d、%s、%x、%c、%p 等格式化字符的处理, 对于在 % 后增加格式要求的格式字符串也要求能够正确识别并输出。这是进行正确测试前必须要做的。具体可以通过一个临时的测试文件进行 debug。

## 3.1.3 task PA2.3: 完善 IOE, 运行 slide、打字小游戏、超级玛丽, 注意跑分情况

必须要正确实现扩展的 IOE, 后面的程序才能运行正确, 各种输入的信息需要使用 inl 指令从端口中获取。viedo.c 中的关于显存像素信息的写入需要严格注意屏幕大小与需要写入图像宽度之间的关系, 不然会导致超级玛丽无法正确显示。

如图 3-1 所示, 这是本次 microbench 跑分情况。



```
khan@khan: ~/ics2018/nexus-am/apps/microbench
File Edit View Search Terminal Help
min time: 1153 ms [478]
[queen] Queen placement: * Passed.
min time: 1561 ms [330]
[bf] Brainf**k interpreter: * Passed.
min time: 8454 ms [310]
[fib] Fibonacci number: * Passed.
min time: 15567 ms [183]
[sieve] Eratosthenes sieve: * Passed.
min time: 13775 ms [307]
[15pz] A* 15-puzzle search: * Passed.
min time: 2929 ms [197]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 2324 ms [582]
[lzip] Lzip compression: * Passed.
min time: 6783 ms [390]
[ssort] Suffix sort: * Passed.
min time: 1372 ms [431]
[md5] MD5 digest: * Passed.
min time: 13049 ms [150]
=====
MicroBench PASS      335 Marks
                   vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 3-1 microbench 跑分情况

# 华中科技大学课程设计报告

如图 3-2、3-3 所示，这是打字游戏、超级玛丽的运行情况。

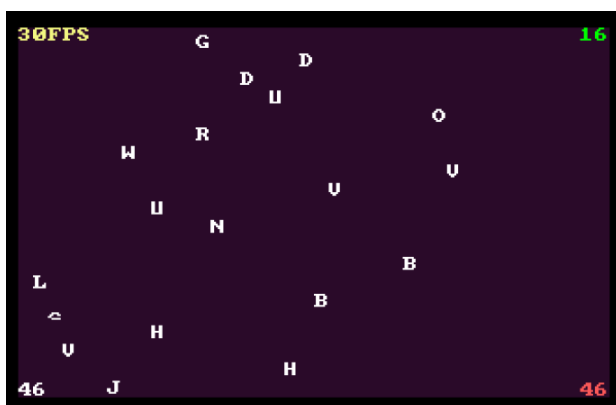


图 3-2 打字游戏



图 3-3 超级玛丽

## 3.2 必答题

1.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？

答：本问题选择了 rtl\_push 函数进行实验。

(1) 去掉 inline，报的错误是函数定义未使用。

(2) 去掉 static，报的错误是内联的非 static 的 push 函数调用了其他 static 函数。

(3) 去掉 inline 和 static，错误是函数重复定义 multiple definition。

static 函数限制了文件的使用范围，静态函数只是在声明他的文件当中可见，不能被其他文件所用，如果非 static 函数调用了 static 函数，就会破坏这种保护机制。所以去掉 static 会报这个错误。具体报错如图 3-4 所示。

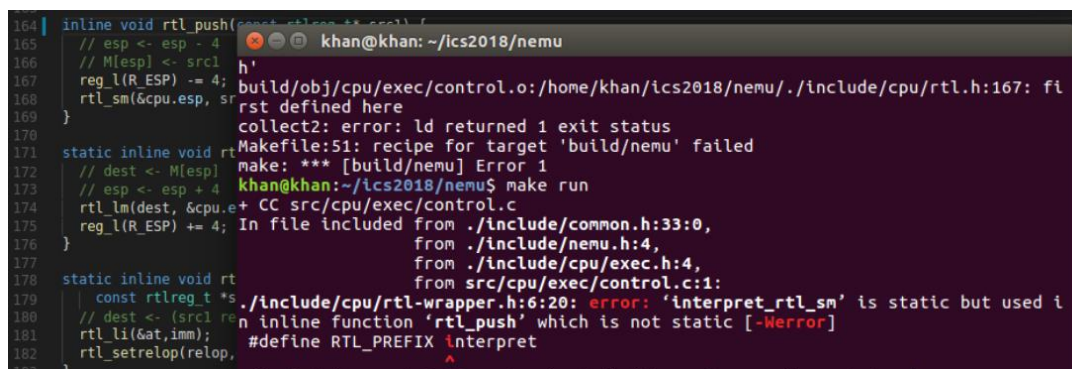


图 3-4 去掉 static 关键字报错

# 华中科技大学课程设计报告

inline 函数代替了宏的功能，避免调用函数时对栈空间的反复占用，static 关键字避免了重复定义的报错，但是编译器会检查每一个 include rtl.h 的文件中有没有使用到对应的 static 函数，这些 static 函数其实都是对应 include 的文件私有的，彼此没有关系，故在一个文件中没有找到这个函数的话就会认为这个函数定义了但是没有使用。使用 inline 关键字会默认函数已经调用过，需要检查函数是否可以内联。具体报错如图 3-5 所示。

```
164 | static void rtl_push(const rtlreg_t* src1) {
165 |     // esp <- esp - 4
166 |     // M[esp] <- src1
167 |     reg_l(R_ESP) -= 4;
168 |     rtl_sm(&cpu.esp, src1);
169 | }
170 |
171 | static inline void rtl_li(const rtlreg_t* src1, rtlreg_t* dest) {
172 |     // dest <- M[esp]
173 |     // esp <- esp + 4
174 |     rtl_lm(dest, &cpu.esp);
175 |     reg_l(R_ESP) += 4;
176 | }
177 |
178 | static inline void rtl_setrelop(relop, rtlreg_t* dest, rtlreg_t* src1) {
179 |     const rtlreg_t* s;
180 |     // dest <- (src1 relop s)
181 |     rtl_li(&at, imm);
182 |     rtl_setrelop(relop, dest, src1);
183 | }
184 | //取出符号位
185 | static inline void rtl_srl(const rtlreg_t* src1, rtlreg_t* dest) {
186 |     // dest <- src1[width-1:0] >> 1
187 |     uint32_t offset = 8;
188 |     uint32_t temp = (*src1) >> offset;
189 |     *dest = temp & 0xf;
190 | }
```

```
khan@khan: ~/ics2018/nemu
./build/nemu -l ./build/nemu-log.txt -d /home/khan/ics2018/nemu/tools/qemu-diff/
build/qemu-so
[src/monitor/monitor.c,54,load_default_img] No image is given. Use the default b
uild-in image.
[src/monitor/monitor.c,33,welcome] Debug: OFF
[src/monitor/monitor.c,36,welcome] Build time: 16:24:38, Dec 28 2018
Welcome to NEMU!
For help, type "help"
(nemu) q
khan@khan:~/ics2018/nemu$ make run
+ CC src/cpu/exec/control.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/system.c
In file included from ./include/cpu/decode.h:6:0,
from ./include/cpu/exec.h:9,
from src/cpu/exec/system.c:1:
./include/cpu/rtl.h:164:13: error: 'rtl_push' defined but not used [-Werror=unus
ed-function]
static void rtl_push(const rtlreg_t* src1) {
Makefile:31: recipe for target 'build/obj/cpu/exec/system.o' failed
make: *** [build/obj/cpu/exec/system.o] Error 1
```

图 3-5 去掉 inline 关键字报错

去掉两者，则该函数定义变成一个普通函数定义，但是该函数在头文件中，会被很多其他文件 include，会造成普通函数的大量重复定义，所以会报错。具体报错如图 3-6 所示。

```
164 | void rtl_push(const rtlreg_t* src1) {
165 |     // esp <- esp - 4
166 |     // M[esp] <- src1
167 |     reg_l(R_ESP) -= 4;
168 |     rtl_sm(&cpu.esp, src1);
169 | }
170 |
171 | static inline void rtl_li(const rtlreg_t* src1, rtlreg_t* dest) {
172 |     // dest <- M[esp]
173 |     // esp <- esp + 4
174 |     rtl_lm(dest, &cpu.esp);
175 |     reg_l(R_ESP) += 4;
176 | }
177 |
178 | static inline void rtl_setrelop(relop, rtlreg_t* dest, rtlreg_t* src1) {
179 |     const rtlreg_t* s;
180 |     // dest <- (src1 relop s)
181 |     rtl_li(&at, imm);
182 |     rtl_setrelop(relop, dest, src1);
183 | }
184 | //取出符号位
185 | static inline void rtl_srl(const rtlreg_t* src1, rtlreg_t* dest) {
186 |     // dest <- src1[width-1:0] >> 1
187 |     uint32_t offset = 8;
188 |     uint32_t temp = (*src1) >> offset;
189 |     *dest = temp & 0xf;
190 | }
```

```
khan@khan: ~/ics2018/nemu
+ CC src/cpu/exec/cc.c
+ CC src/cpu/exec/special.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/arith.c
+ CC src/cpu/exec/logic.c
+ CC src/cpu/decode/decode.c
+ CC src/cpu/decode/modrm.c
+ CC src/cpu/intr.c
+ LD build/nemu
build/obj/cpu/exec/data-mov.o: In function 'rtl_push':
/home/khan/ics2018/nemu/./include/cpu/rtl.h:167: multiple definition of 'rtl_pus
h'
build/obj/cpu/exec/control.o: /home/khan/ics2018/nemu/./include/cpu/rtl.h:167: fi
rst defined here
build/obj/cpu/exec/system.o: In function 'rtl_push':
/home/khan/ics2018/nemu/./include/cpu/rtl.h:167: multiple definition of 'rtl_pus
h'
build/obj/cpu/exec/control.o: /home/khan/ics2018/nemu/./include/cpu/rtl.h:167: fi
rst defined here
```

图 3-6 去掉 static、inline 关键字报错

## 2. 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



# 华中科技大学课程设计报告

个结果的?

答: 重新编译后的 NEMU 含有 30 个 dummy 的实体。

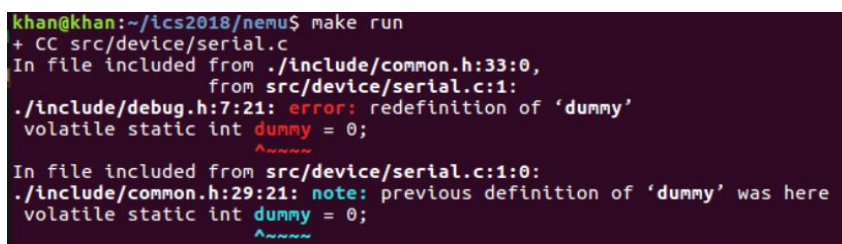
流程: 在未添加任何代码时, 对 nemu 的可执行文件进行反汇编, 反汇编只有 dummy 程序相关的几个 dummy 关键词, 没有变量的实体。修改代码运行 nemu 后, 对 nemu 的可执行文件进行反汇编, 观察到在 bss 段处有 30 个 dummy 变量的实体, 在采用段式内存管理的架构中, BSS 段通常是指用来存放程序中未初始化的全局变量的一块内存区域, 属于静态内存分配。

(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。

答: 再添加一行代码编译 NEMU 仍然只有 30 个 dummy 变量的实体。因为 common.h 中同时也 include 了 debug.h, debug.h 中也 include 了 common.h, 所以这两句声明相当于写在同一个文件中。并且 c 语言编译器对一模一样的变量声明不会报错, 只有类型不一样或者重复定义才会报错。

(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?

答: 报错: redefinition of 'dummy' 以及 previous definition of 'dummy' 的错误出现, 根据报错很明显可以得知, 无论在什么地方, 同一个项目中变量是不可以重复定义的。之前只是声明变量, 而且类型完全相同, 编译器会忽略这种情况, 当作只有一个变量声明了。报错情况如图 3-7 所示。



```
khan@khan:~/ics2018/nemu$ make run
+ CC src/device/serial.c
In file included from ./include/common.h:33:0,
                 from src/device/serial.c:1:
./include/debug.h:7:21: error: redefinition of 'dummy'
volatile static int dummy = 0;
                    ^~~~~~
In file included from src/device/serial.c:1:0:
./include/common.h:29:21: note: previous definition of 'dummy' was here
volatile static int dummy = 0;
                    ^~~~~~
```

图 3-7 初始化后报错情况

3. 了解 Makefile 请描述你在 nemu/目录下敲入 make 后, make 程序如何组织.c 和.h 文件, 最终生成可执行文件 nemu/build/nemu。

答: Makefile 里面用.PHONY 明确声明几个用于执行的伪目标, 重写了一些模糊的规则, 使得 make run、make gdb、make clean、make app 执行特定的指令。在 makefile 中生成可执行文件的伪目标是 app, make 默认执行这个目标。run 伪目标中包含了 app

伪目标所需要执行的指令而已。

设置系统变量 CC 使用 gcc 编译器编译, 设置 LD 使用 gcc 链接, 通过自定义变量 INCLUDE 与 CFLAGS 以及其内部使用到的一些指明路径的变量, 构建了一条链接 c 程序 and 对应头文件 and 开启了一系列检查标志具有特定编译格式的 shell 命令。然后 Binary 变量中调用了这条指令, 从而完成了编译链接, 最终生成可执行文件。

## 3.3 主要故障与调试

### 3.3.1 初次运行 dummy 系统提示找不到 sys/cdefs.h

出错原因: 系统缺少对应的包

解决方案: 安装 libc6-dev-i384 这个包即可解决问题。

### 3.3.2 算术右移错误

出错原因: 算术右移未考虑操作数宽度。

解决方案: 对不同宽度的操作数都严格执行算术右移。

### 3.3.3 跑 coremark 时分数输出错误

出错原因: 检查了 printf 表达式中单个元素都没问题, 只有合起来有问题, 汇编指令采用的执行函数错误。

解决方案: 检查到 imul 指令中使用到三个操作数的情况下错误的使用了 imul2 函数, 使得计算结果错误, 将其改用为 imul3 来执行即解决问题。

### 3.3.4 Videotest 正常, 之后的超级玛丽、slider、打字游戏不正常

出错原因: 实现 video\_write 函数时忽略了所画矩形超出屏幕宽度的情况。

解决方案: 在实现 video\_write 函数加入对边界的限制, 即可实现所有测试程序的正常运行。

### 3.3.5 Diffptest 突然找不到 qemu 的句柄 handle

出错原因: make clean 会清除掉编译好的 qemu-so。

解决方案: 进入 nemu/tools/qemu 重新编译 qemu。



## 4 PA3 批处理系统

### 4.1 功能实现的要点

#### 4.1.1 task PA3.1: 实现自陷操作\_yield()及其过程。

重要知识要点，IDT 寄存器和 IDT 表的结构，以及利用 IDTR 对 IDT Table 进行索引的方式具体如图 4-1、图 4-2 所示。IDTR 是一个 6 个字节的寄存器，分为两个部分，一个是 base 部分——在高位占 4 个字节，一个 limit 部分——在低位占 2 个字节。在实现 lidt 指令时，需要根据指令中提供的地址，按照 IDTR 的结构读出 6 个字节的信息，然后将其放入 cpu 结构体的 IDTR 中。

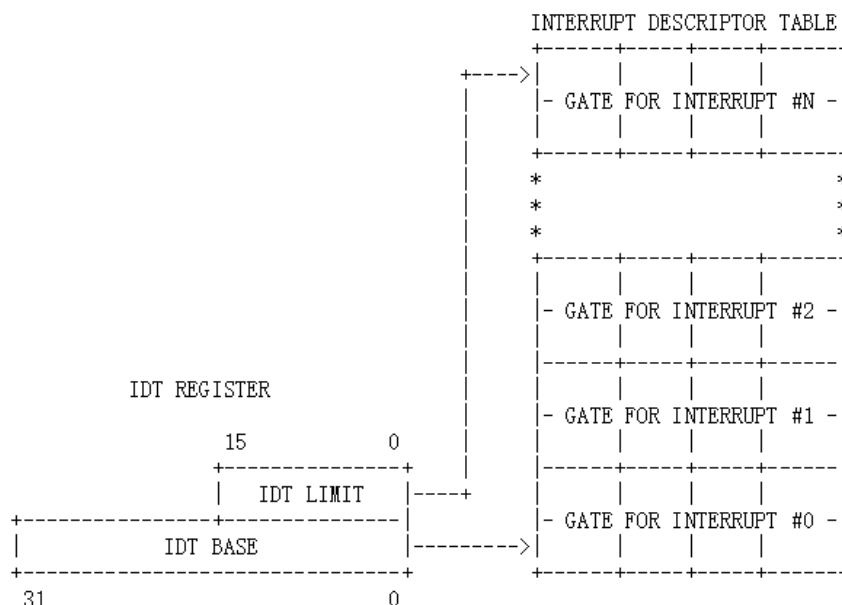


图 4-1 IDT 表的结构及其索引方式

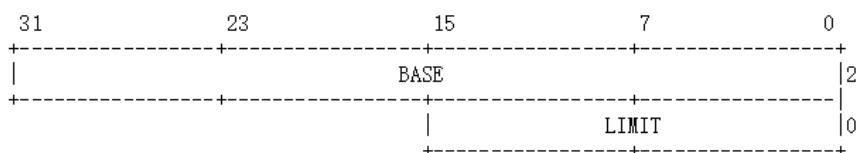


图 4-2 IDT 寄存器具体结构

实现 int、iret、popa、pusha 指令，其中需要注意 int 和 iret、pusha 和 popa 中各自 push 和 pop 的内容是一一对应的。

对于 int 指令，最重要的是取出门描述符，对其内容进行解析，解析出地址后进行跳转，如图 4-3 所示，这是门描述符的结构，需要根据 p 位组合高位和低位。

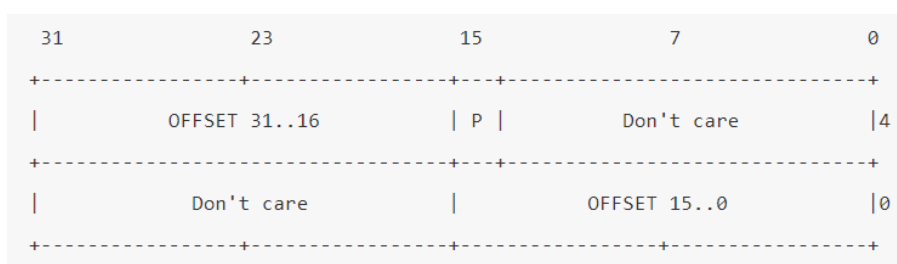


图 4-3 门描述符的结构

## 4.1.2 task PA3.2: 实现用户程序的加载和系统功能调用, 支撑 TRM 程序的运行

根据 `syscall` 的系统功能调用对返回值和参数的绑定, 需要对 `GPRn` ( $n=1, 2, 3, 4, 5$ ) 宏进行正确的设置, `GPR1` 代表 `eax`、`GPR2` 代表 `ebx`、`GPR3` 代表 `ecx`、`GPR4` 代表 `edx`、`GPR5` 代表 `eip`。`eax` 是函数返回值存放的寄存器, 而 `ebx`、`ecx`、`edx` 存放的是传入函数的参数, 依照函数参数具体数目使用它们。

添加系统功能调用首先要在 `IDTtable` 下标 `0x80` 处添加系统功能调用门描述符相关内容, 才能够正常使用, 对于系统自陷的门描述符的填充已经有了相关的实现。然后在 `navy-apps/libs/libos/src/nanos.c` 中进行系统功能调用函数的绑定, 然后在 `do_syscall` 函数中根据上述的 `GPR` 宏, 读取参数, 确认系统功能调用的类型, 设置返回值, 才能正确添加系统功能调用。

其中 `SYS_brk` 中维护的 `program break` 是在用户态下维护的一个值, 需要利用程序的 `&_end` 参数进行初始化, `&_end` 在程序过程不会变化, 所以需要自己维护一个 `program break`。

为了实现系统功能调用, 本次编写了 `fs_open`、`fs_close`、`fs_read`、`fs_write`、`fs_lseek` 等函数, 在对应的系统功能调用中进行调用。在实现这几个函数的过程中, 需要使用到 `Finfo` 结构体, 里面的参数有文件名、`size` 文件大小、`disk_offset` 文件相对于 `ramdisk` 的偏移、`open_offset` 文件内的已读取\写入的偏移、还有关于控制文件读写的函数指针, 这是对文件的一个抽象, 有一点 OOP 的思想, 函数指针类似方法, 其他成员是属性, 通过使用函数指针可以实现文件的抽象。`read`、`write` 需要时刻维护 `open_offset`, 注意文件大小与读取宽度之间的关系。

对于 `stdin`、`stdout`、`stderr`、`/dev/events`、`/dev/tty` 等不设置文件大小的文件不需要检查大小, 直接调用读写接口即可。如图 4-4 所示, 这是实现 `/dev/events` 后实现的事件记录效果。

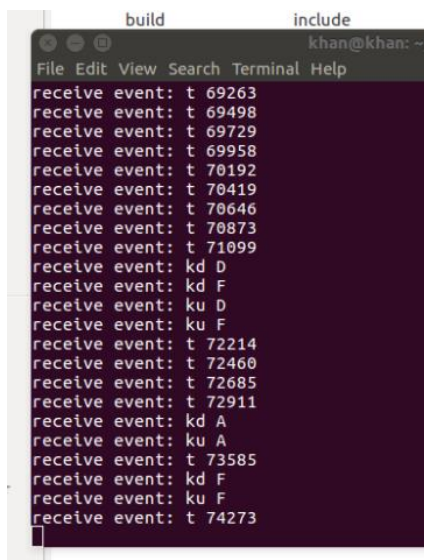


图 4-4 时间事件和键盘事件的识别结果

### 4.1.3 task PA3.3: 运行仙剑奇侠传展示批处理系统

按照讲义进行实验即可。注意文件名的书写和记得在改变了 nanos 文件的时候要 make ARCH=x86-nemu update, 再重新开始运行。

如图 4-5 所示, 这是在批处理系统下运行的仙剑奇侠传。

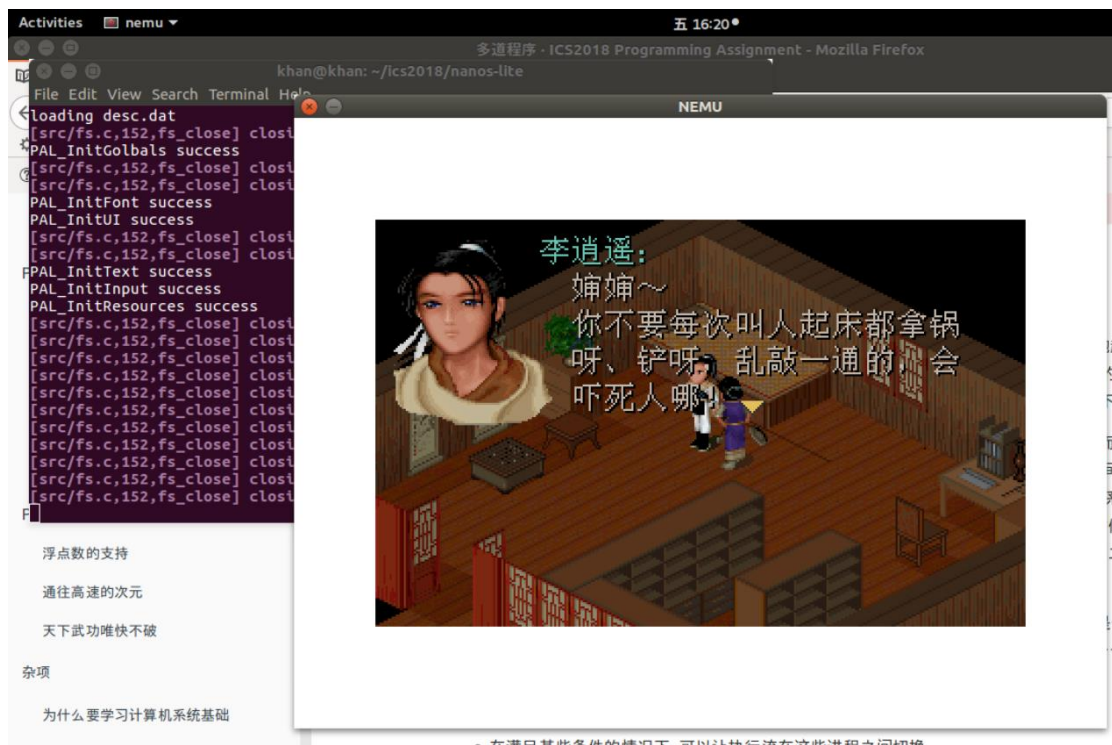


图 4-5 在批处理系统下运行的仙剑奇侠传

## 4.2 必答题

1.文件读写的具体过程 仙剑奇侠传中有以下行为:

(1) 在 navy-apps/apps/pal/src/global/global.c 的 PAL\_LoadGame()中通过 fread()读取游戏存档;

(2) 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.

答: 库函数, libos, Nanos-lite, AM, NEMU 的层次是由高到低的, NEMU 提供了硬件, AM 在 NEMU 之上提供了运行时环境, Nanos-lite 作为操作系统, 运行在了架构为新-nemu 的 AM 之上, libos 中实现了操作系统中系统功能调用, 而库函数是用户方便用户调用的函数, 省去很多编写程序的麻烦, 用户程序可以使用库函数和库函数内部或者用户调用系统功能调用, 在用户态和核态下完成对应应该完成的工作。上述讲的是库函数, libos, Nanos-lite, AM, NEMU 总体的相互协助的逻辑。

具体到仙剑奇侠传中的游戏存档的读取,使用了库函数 fopen、fclose、fread, 这些库函数中再使用系统功能调用, 对给定文件名的文件进行打开、读取、关闭。读取出来字节流后, 读取函数对字节序进行了修改, 然后读取出对应字段的信息出来。

至于屏幕的更新,使用的是NDL多媒体库,redraw函数调用库函数NDL\_DrawRect对画布进行重绘,所谓重绘,就是在对于的画布像素存储区覆盖上新的像素信息。

这两个功能主要是在软件层面上实现的,使用了库函数及其内部的系统功能调用,而操作系统、AM、NEMU 都成为支持游戏正确运行的基础。

## 4.3 主要故障与调试

### 4.3.1 系统功能调用失败

出错原因: 没有在 IDT table 注册对应的门描述符。

解决方案: 在 IDT table 的 0x80 处添加系统功能调用需要用到的门描述符。

### 4.3.2 fs\_read、fs\_write 函数越界

出错原因: 没有对函数传入的读取\写入长度进行处理,导致读取范围超过文件的大小。

解决方案: 通过文件的大小 size 和此时已经读取\写入的偏移 open\_offset 计算文件剩

余的大小，并将其与函数传入的长度进行比较，两者取较小的一个作为读取\写入的长度，从而使得读取\写入不会超过文件的大小。

## 4.3.3 在运行/bin/bmptest 提示越界

出错原因：在设置/dev/fb 大小时设置错误。

解决方案：通过 IOE 接口函数 screen\_width、screen\_height 获取屏幕宽度，相乘得到的是屏幕的像素数，而一个像素有 RGBA 四个字节，因此对于显存文件的大小需要在像素数的基础上乘以 4，才是正确的文件大小。

## 4.3.4 stderr、stdout 没有正常输出

出错原因：对于 stdin、stdout、stderr、/dev/events、/dev/tty 等文件被 read、write 函数本身读取长度和文件大小的限制控制到了，导致调用底层函数时读取长度被置为 0。

解决方案：它们的文件大小为 0，意思是不做限制，因此需要在对应的读取和写入函数利用文件标识符区分开来处理，对他们不检查文件大小，直接调用读取或者写入函数进行处理。

## 4.3.5 仙剑奇侠传在 x86-nemu 运行正常，在 native 运行失败

出错原因：nanos 文件中关于 open 的系统功能调用加入了多余的判断限制。

解决方案：需要删除对 nanos 文件中对于的判断，然后 make ARCH=x86-nemu update，重新运行。除此之外还发现在 serial\_write 中没有设置正确的返回值，导致输出错误。

## 5 实验总结与心得

本次 pa 实验学习到了很多以前没有使用过的工具，也熟悉了一些以前不够熟悉的工具，本次 pa 完成了 pa0、pa1、pa2、pa3，还有 pa4 没有完成。

Pa1 中主要完成的是正则表达式书写和表达式解析计算，有一种编译原理简化版实验的感觉，复习了正则表达式的写法，以及了解了解析表达式的原理。

Pa2 中主要完成了指令的填写，其中大量查阅了 x86 手册，感觉像是一个汇编与高级程序语言结合的实验，在顶层模拟一条一条指令的执行过程，操作数的计算与转移，标志寄存器的改变等等。同时也学会了利用宏规范化写法，以及程序需要解耦的思想。同时通过自己编写一些库函数，特别是 `printf` 这一族的函数的编写，熟悉了格式化字符串解析的流程。

Pa3 中主要进行操作系统层面的工作，通过进一步完善指令集合，组织系统自陷和编写系统功能调用，完成文件的开关读写、索引，批处理系统等的功能，从而组织各个程序运行，实现游戏程序运行的可能，运行仙剑奇侠传游戏。

总之 Pa 培养了我们的系统能力，虽然没有在规定的时间内完成所有的部分，但是我会在课余时间争取通关。

• 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：刘科翰

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：\_\_\_\_\_