

华中科技大学

2019

系统能力综合训练 课程设计报告

题 目: x86 模拟器开发

专 业: 计算机科学与技术

班 级: CS1706

学 号: U201714761

姓 名: 胡澳

电 话: 17371232678

邮 件: 1186767079@qq.com

完成日期: 2020 年 12 月 31 日

目 录

1	课程设计概述	3
1.1	课设目的	3
1.2	课设任务	3
1.3	实验环境	3
1.4	实验过程概述	3
2	PA0 世界诞生的前夜：开发环境配置	5
3	PA1 开天辟地的篇章：最简单的计算机	7
3.1	简单调试器	7
3.2	表达式求值	8
3.3	监视点与断点	9
3.4	运行结果	9
3.5	问题解答	10
4	PA2 简单复杂的机器：冯诺依曼计算机系统	12
4.1	基本指令的实现	12
4.2	常用库函数的实现	13
4.3	QEMU-DIFF 的实现	13
4.4	IO 指令及功能的实现	15
4.5	运行结果	16
4.6	问题解答	17
5	PA3 穿越时空的旅程：批处理系统	19
5.1	上下文管理	19
5.2	用户程序、系统调用、文件系统	20
5.3	运行结果	21

华中科技大学课程设计报告

5.4	问题解答.....	21
6	PA4 虚实交错的魔法: 分时多任务.....	23
6.1	多道程序.....	23
6.2	分时多任务.....	24
6.3	运行结果.....	24
6.4	问题解答.....	24
7	总结与心得.....	26
7.1	课设总结.....	26
7.2	课设心得.....	26
	参考文献.....	28

1 课程设计概述

1.1 课设目的

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。本次课程设计通过实现一个经过简化但功能完备的 x86 模拟器 NEMU，最终在 NEMU 上运行游戏“仙剑奇侠传”，来探究“程序在计算机上运行”的基本原理。

1.2 课设任务

本次课程设计主要包含下列实验内容。

1. 图灵机与简易调试器
2. 冯诺依曼计算机系统
3. 批处理系统
4. 分时多任务

1.3 实验环境

课设内容在 Arch Linux 上完成，由于该发行版在软件包更新上秉持“尽可能保证软件处于最新的稳定版本”的原则，因此课设中使用到的开发环境及工具链几乎均为使用时的最新版本，下面给出完成报告时的部分开发环境参数。

操作系统：Arch Linux

操作系统内核：x86_64 Linux 5.9.14

桌面环境：Gnome 3.38.2

Xorg: X.Org X Server 1.20.10 / X Protocol Version 11, Revision 0

编译/调试环境：gcc 10.2.0 / gdb 10.1 / GNU Make 4.3

qemu: QEMU emulator version 5.2.0

1.4 实验过程概述

本次课程设计的主要内容从 10 月 2 日开始进行，在接下来的两周时间内基本完成了 PA1/2/3 的内容，由于各种原因，PA4 的内容则在接下来的差不多一个月的时间内零零散散的完成。而实验报告的编写则是在 12 月才开始进行。

华中科技大学课程设计报告

前三次实验内容的提交是在完成 PA4 的过程中一起完成的，在提交时，由于遇到了一个难以解决的关于 qemu 的 bug(该 bug 在后来得到了解决，详见第 2 章中的相关描述)，因此彼时在代码中将 qemu-diff 功能关闭，同时解决了一些小 bug 后进行了提交，因此导致实际提交的内容在 git 树上表现为 master 主分支以外的几个小分支。

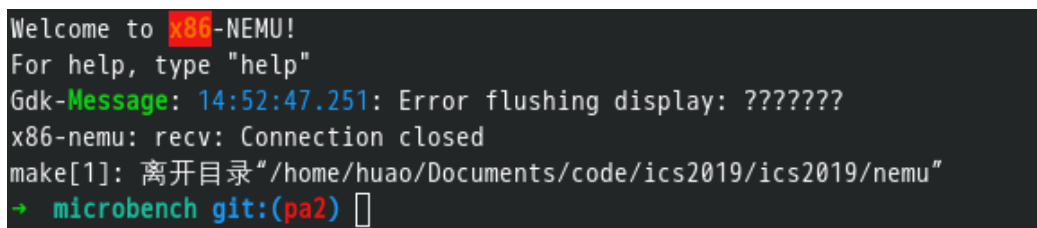
在编写报告时，我觉得在上述提交方式有些不妥，因此决定将前面进行的修改转移到 PA4 完成后的位置，从而保证其位于主分支 master 上，修改完成后，对所有实验内容进行重新提交。这样，虽然最终的 PA1/PA2 分支头的位置被转移到了 PA4 的前面，但提交的每个点都汇聚到了 master 分支中的某个位置，我认为这样要更好一些。在后续的测试中，若无特殊需求(例如，PA3 中运行仙剑奇侠传游戏时，若在实现了虚存的机器上运行的话，其速度很慢，因此我们切换到 PA3 上运行)，我们均可以直接在 master 分支的位置进行测试。

2 PA0 世界诞生的前夜: 开发环境配置

PA0 的主要任务为安装和配置实验环境,由于我们在其他课程实验中使用过 linux 操作系统,因此在本次课程设计中我们直接使用了原有的操作系统进行,相关的操作系统及软件环境在 1.3 节中已经给出了较为详细的描述。

此处我们描述一个实验中遇到的一个与实验环境相关的问题,虽然该问题在后续的 PA 中才会遇到,但由于导致该问题的原因是实验环境,因此我们在这里对问题及其解决方案进行描述。

如图 2-1 所示,在启用 qemu-diff 功能后, qemu 运行一段时间后报错退出。



```
Welcome to x86-NEMU!  
For help, type "help"  
Gdk-Message: 14:52:47.251: Error flushing display: ???????  
x86-nemu: recv: Connection closed  
make[1]: 离开目录 "/home/huao/Documents/code/ics2019/ics2019/nemu"  
-> microbench git:(pa2) [ ]
```

图 2-1 qemu 运行错误

为解决上述问题,使 qemu-diff 功能能够正常工作,我们进行了如下调试工作。首先,在关闭 qemu-diff 功能时,程序可以正常运行;其次,开启 debug 模式记录虚拟机执行过的每一条指令,多次运行同一程序后可以发现,在 qemu 报错退出前,系统已经开始运行,即 nemu 和 qemu 都执行了部分指令且完成了正确性比较,但每次 qemu 报错退出时系统已经执行的指令数是不同的。根据上述运行结果,我们可以初步断定,该故障出现的流程如下。

系统正常启动了 qemu,并将其与 nemu 进行连接,此后,二者同步执行了一些指令并进行寄存器比较,在 qemu 运行的某个时刻出现错误导致 qemu 退出、qemu 与 nemu 之间的连接中断,当 nemu 执行完一条指令后尝试从 qemu 读取寄存器状态进行比较时发现连接断开导致系统异常终止。

由于其中导致 qemu 出错的发生具有一定的随机性,且报错信息表示错误是 qemu 刷新显示器出错,因此我们初步认定该问题可能是由于开发环境导致的。

我们尝试对 qemu 等软件进行降级、重装操作系统等措施,在机缘巧合之下,我发现,在使用 KDE 而不是 Gnome 作为桌面环境时,该问题不再出现。结合我在 Arch

华中科技大学课程设计报告

Linux 文档中看到的关于 Gnome 的介绍,我猜测该问题是由于二者默认使用的窗口系统不同, KDE 默认使用较为传统的 Xorg, 而较新版本的 Gnome 虽然也包括并安装了 Xorg, 但默认使用 Wayland 而不是 Xorg, 而课程设计的框架代码或者 qemu 或许使用了一些与 Xorg 强相关的功能, 导致上前面发生的问题。最后, 我们使用 Gnome on Xorg 作为桌面系统, 终于正确运行了 qemu-diff 功能。

3 PA1 开天辟地的篇章: 最简单的计算机

3.1 简单调试器

为了便于后期对 `nemu` 的调试, 在开始实现虚拟机之前, 我们需要为 `nemu` 实现一些简单的调试功能, 例如单步执行、打印程序状态、扫描内存、表达式求值以及程序断点等。其中, 表达式求值涉及较为复杂的算法, 程序断点需要增加一系列子功能, 因此这两个调试功能我们分别在 3.2 和 3.3 节中单独描述, 在这一节中, 我们主要介绍其余几项调试功能的设计与实现。

3.1.1 单步执行 `si`

通过阅读框架代码, 我们可以得知, 在 `nemu` 中使用函数 `exec_once` 来完成对下一条指令的执行并更新机器状态, 因此单步执行的功能通过直接调用该函数即可实现。另外, 单步执行可以接受一个整数参数, 表示向前执行多条指令, 此功能仅需要循环调用 `exec_once` 函数多次即可实现。

3.1.2 打印程序状态 `info`

程序状态包括两种类型, 寄存器状态和监视点状态。后者我们在 3.3 节监视点与断点中再做说明, 此处首先完成寄存器状态的打印。

所谓寄存器状态, 其实就是各寄存器的值, 由于寄存器的数量、名称等与指令集相关, 因此我们将输出寄存器状态的功能封装成指令集无关的函数 `isa_reg_display`, 该函数在指令集相关的源文件中进行具体的实现。对于 `x86` 指令集, 我们逐个输出其 8 个通用寄存器以及 `PC` 寄存器的值即可。最后在 `info` 功能中调用 `isa_reg_display` 函数即可实现寄存器状态的输出。

3.1.3 扫描内存 `x`

在 `nemu` 中, 我们直接使用一个大数组 `pmem` 对内存进行模拟, 因此内存扫描实际上就是读取该数组中的部分内容进行输出。从输入参数中读取到内存地址以及需要扫描的内存字节数, 从 `pmem` 中取出这些内容输出即可。

3.2 表达式求值

表达式求值功能是根据输入的表达式字符串计算其结果，因此该功能需要分两步完成，字符串解析和表达式求值。

3.2.1 表达式字符串解析

在表达式字符串中，存在加减乘除等运算符、十进制/十六进制数字、等于/不等于等逻辑运算符、空格以及寄存器标识符等符号，我们使用正则表达式对其进行逐个的匹配，最终将字符串解析成一个符号列表。比较特殊的是，对于表达式中解析出的寄存器标识符，我们可以在此处从机器中读出相应的数值，从而作为数字直接参与后续的计算。

3.2.2 表达式求值

完成字符串解析后，我们的表达式求值工作将在一个符号列表上进行。为便于描述求值过程中使用的算法，下面我们仅考虑表达式符合逻辑的情况。

首先，我们考虑表达式中可能出现的括号，对表达式符号列表从前向后扫描，当我们遇到一个左括号的时候，在其后必定存在且仅有一个与之匹配的右括号，而这两个括号之间的子符号列表则是一个子表达式，我们可以对其进行递归调用表达式求值函数，求出子表达式的结果以后，包围该子表达式的括号对自然就可以去掉了。

经过上述操作，我们最终可以得到一个不包含括号、仅包含数字与运算符的表达式。对于此类表达式的计算，我们采用编译原理课程中讲述的相关算法进行处理。由于我们假定表达式合法，因此表达式的结构必定是数字与运算符交替出现。我们建立一个符号栈，交替从表达式符号列表中解析出数字和运算符，放入符号栈中。对于数字，由于表达式中可能出现冗余的正负号，因此从符号列表当前位置开始逐个向后扫描，直到找到一个数字，同时统计其前面紧跟的‘*’符号的数量以及‘-’符号的数量，根据前者，我们对数字进行多次的地址解析，对于后者则决定了该数字的正负。对于运算符，我们仅需要从符号列表中取出最前面的符号即可。

考虑到乘除法 > 加减法 > 逻辑运算的优先级关系，我们对完全解析后的符号栈进行三次计算，分别处理掉上述三种类型的运算符即可完成表达式的求值。以乘除法为例，当我们遇到一个乘号时，将其前后的两个数字相乘得到的结果替换掉这两个数

字以及中间的乘号即可。这样的操作，我们可以使用双向队列很轻易的实现。

处理完三种不同的运算符后，我们最终可以得到表达式的运算结果。

当然，实际的运算过程中，我们可能会遇到表达式本身存在错误的情况，当我们遇到不能继续执行下去的符号时，根据符号结构体中保存的该位置相对于表达式首部的位置，给出相应的错误信息提示，保存运算错误状态并退出计算即可。

3.3 监视点与断点

对于监视点功能，我们需要实现四个相关的功能，监视点的创建、删除、输出以及检查。我们使用两个单链表来保存系统中监视点的信息，分别为已使用的监视点链表 `head` 和可使用的监视点链表 `free_`，每个监视点包含编号、监视表达式字符串以及上一次表达式计算的结果。

对于监视点的创建，我们从 `free_` 链表中取出一个空监视点，将其插入到 `head` 链表中，为了使创建能够在 $O(1)$ 的时间内完成，我们可以直接将新监视点插入到 `head` 链表的头部。若 `free_` 链表为空，即没有多余的空监视点可以使用了，我们不出来这种情况，直接给出相应的报错信息。

对于监视点的删除，我们遍历 `head` 链表，找到待删除的监视点，将其从链表中移除并放回到 `free_` 链表中即可。若未能找到该监视点，那么我们不对监视点链表做任何操作。

对于监视点的输出，我们遍历 `head` 链表，逐个输出监视点的编号及表达式即可。

最后，监视点的检查在每次执行完一条指令后进行，即在 `exec_once` 函数中，完成原有操作后，调用监视点检查的函数，若某个监视点监视的表达式的值发生了变化，那么将虚拟机状态切换为 `NEMU_STOP` 状态，并输出该监视点的信息。至于监视点检查函数，其遍历所有的已使用监视点，调用 3.2 节中实现的表达式求值功能对监视表达式进行求值，若表达式的值发生变化，那么更新该监视点并将其返回。

3.4 运行结果

根据老师提供的测试数据，我们对 `nemu` 的调试器进行测试，如图 3-1 为对单步运行、查看程序状态、检查点等功能的测试，如图 3-2 为对表达式求值功能的测试。通过测试结果可知，简单调试器的功能已基本实现。

华中科技大学课程设计报告

```
Welcome to NEMU!
For help, type "help"
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Step N instructions, default one instruction
info - Print program status. 'r' for register, 'w' for watchpoint
x - Scan memory. Use 'x N EXPR' to print 4*N bytes from addr EXPR
p - Calculate expr.
w - Add a watchpoint.
d - Delete a watchpoint.
attach - attach diff_test with qemu.
detach - detach diff_test with qemu.
save - save nemu status.
load - load nemu status from file.
(nemu) si
$EPC: 00100005
(nemu) si 2
$EPC: 0010000c
(nemu) info r
EAX      00001234          4660          EBX      58c002d4          1488978644
ECX      00100027          1048615         EDX      4db90a4b          1303972427
ESP      52f3d485          1391711365        EBP      3d73b6ea          1030993642
ESI      6cab82fb          1823179515        EDI      5afb0fac          1526460588
EIP      0010000c
CF:0   ZF:0   SF:0   OF:0
(nemu) w $eip==0x100005
watchpoint 0: $eip==0x100005
(nemu) w $eax
watchpoint 1: $eax
(nemu) w $ecx
watchpoint 2: $ecx
(nemu) d 2
(nemu) info w
watchpoint 1: $eax
watchpoint 0: $eip==0x100005
(nemu) 
```

图 3-1 调试功能测试

```
(nemu) p (1+2)*(4/3)
3
(nemu) p (3/3)+(123*4
unexpected right bracket at position 6
(3/3)+(123*4
^
expr error
(nemu) p (1+(3*2) +(($eax-$eax) +(*$eip-1*$eip)+(0x5--5+ *0X100005 - *0x100005) )*4)
47
(nemu) 
```

图 3-2 表达式求值功能测试

3.5 问题解答

这里我们来对实验文档中的必答题进行逐一的解答。

1. 我选择的 ISA 是 x86。

2. 按照题目假设的数据，如果不实现简单调试器，那么花在调试上的时间为 7.5 小时，实现简单调试器后的调试时间将降低为 2.5 小时，由此可见，通过实现一定的基础设施，可以有效的减少我们在后续工作中 debug 的工作量。

3. eflags 寄存器中的 cf 位为进位位，我们曾在汇编、组原等课程中详细讨论过计算过程中的进位问题。

华中科技大学课程设计报告

ModR/M 字节是 x86 中与操作数寻址方式相关的标识位。

x86 中的 `mov` 指令包含了一系列的指令，它们适用于不同的寻址方式、不同的操作数位数，在指令集文档中有详尽的说明。

4. 这是一个关于代码行数的问题。我认为关心自己写了多少行代码是一件没什么意义的事情，与其关心这个，不如想想自己的代码是否写得简明易懂，有没有好好写注释和文档，而不是说为了把代码量变大而写入大量的废话，亦或是为了把代码量变小而使用大量炫技的编码技巧导致代码变得晦涩难懂。

5. gcc 的编译选项 `-Wall` 表示显示编译过程中所有的警告 `warning`，而 `-Werror` 表示将所有的警告视为错误来处理。这样做的目的应该是为了避免一些潜在的 `bug`，即使 gcc 通常认为只是一个 `warning`，我们仍然需要将其改正。

4 PA2 简单复杂的机器：冯诺依曼计算机系统

4.1 基本指令的实现

本次课程设计中，我选择实现 x86 指令集的 nemu 虚拟机。虚拟机 nemu 最基本的功能就是对程序指令的执行。一条指令的执行可以分为以下四个步骤，取指、译码、执行、更新 PC。在框架代码中，已经实现了取指和更新 PC 的操作，并给出了译码和执行的流程，我们只需要根据指令集填写操作码表 `opcode_table` 并实现相应指令的译码和执行函数就可以实现指令。

由于 x86 为变长指令集，我们需要逐字节的读取操作码以判断指令类型从而决定如何继续对指令中的操作数进行译码，具体流程如图 4-1 所示。其中操作数译码对于不同指令的操作是相同的，绝大多数指令的译码函数均已在框架代码中给出，可以直接填入操作码表中。

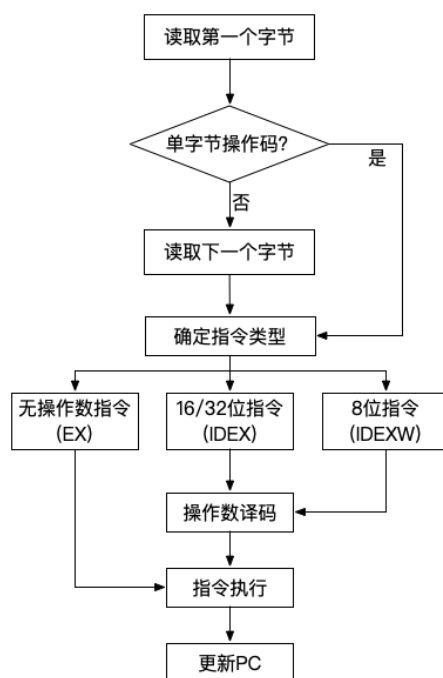


图 4-1 指令执行流程

由于 x86 包含大量的指令，因此我们并不一次性根据指令集将所有的指令全部在 nemu 中进行实现，而是随着课程设计的需要，不断地为虚拟机增加新指令的支持。另外，考虑到 x86 中相同功能的指令往往存在一组，以支持不同位宽的操作数以及操作数不同的寻址方式，这一组指令往往使用相同的执行函数，因此当我们遇到其中一

个指令时，可以同时完成这一组中的其他指令，从而避免为了同组指令多次查阅指令集手册。

结合框架代码中提供的测试程序，我们向 `nemu` 中增加允许所需的指令支持，最终使得除与字符串相关的程序外的所有的测试程序都能够正常的执行到“HIT GOOD TRAP”的位置。

4.2 常用库函数的实现

通过实现各种指令，我们已经能够通过除 `string.c` 和 `hello-str.c` 外的全部 `cputest` 测试，为了通过余下的两个测试程序，我们需要实现一些字符串/内存相关的库函数，包括字符串长度、拷贝、连接、比较，以及内存的赋值、拷贝和比较等库函数。这些功能较为基础，一般可通过一次遍历即可实现，此处不做过多描述。

另外，我们还需要实现字符串的格式化处理 `sprintf` 功能，由于 `sprintf` 与 `printf` 功能类似，区别仅在于前者将格式化的字符串输出到指定内存位置，而后者输出到标准输出，因此我们将字符串的格式化在 `vsprintf` 函数中加以实现，在 `sprintf/printf` 中直接调用 `vsprintf` 即可实现两种不同的格式化输出。下面我们对 `vsprintf` 函数的实现进行具体的解释。

函数 `vsprintf` 接受三个参数，分别为输出字符串、格式字符串以及参数列表。其流程如下。对格式字符串进行遍历，对于非‘%’的字符，我们直接原样添加到输出字符串末尾，对于‘%’字符，表示接下来我们需要对参数进行输出，此时我们读取下一个字符，已确定对参数的输出方式。在 C 标准库中，`printf` 支持的格式化方式很多，此处我们仅实现一些后续需要使用到的功能，例如 `%s` 输出字符串、`%d` 输出十进制整数以及 `%0nd` (`n` 为一位十进制整数) 输出带前序 0 的十进制整数。对于输出字符串，我们直接将参数字符串的内容追加到输出字符串的末尾即可，对于十进制数字的输出，我们采用除十取余的方式即可实现。

完成了上述字符串/内存相关的基本库函数后，我们运行两个与之相关的测试程序，它们可以正常运行至“HIT GOOD TRAP”的位置。至此，我们通过了全部的 `cputest` 测试。

4.3 qemu-diff 的实现

事实上，测试程序能够执行到指定位置并不能够完全说明指令实现的正确性，程

华中科技大学课程设计报告

序仍然可能存在一些潜在的 bug，而这些 bug 并没有对测试程序的执行流程产生太大的影响。为了真正验证指令实现的正确性，我们需要使用 diff-test 功能，将我们实现的 nemu 运行过程中的寄存器状态与我们认为正确的虚拟机 qemu 的运行状态进行比较，若每条指令执行完成后二者的状态均相同，我们才能够真正认为指令的实现是正确的。

在框架代码中已经给出了 qemu-diff 的具体流程，每条指令执行完成后，对 nemu 和 qemu 中的寄存器状态进行比较，由于我们已经按照标准对寄存器结构体中的各成员变量进行了排列，因此此处的比较可以直接调用前面实现的内存比较函数进行比较，若比较结果为二者相同，则认为当前指令执行没有问题，否则，表示指令实现存在错误，我们将当前 nemu 和 qemu 的寄存器内容输出，并将 nemu 虚拟机的状态置为 NEMU_ABORT，从而终止虚拟机的运行。

完成上述操作后，我们就可以对 cputest 中的测试样例进行回归测试，对出现问题的指令加以修正，最终我们可以通过全部的测试样例，如图 4-2 所示。

```
NEMU compile OK
compiling testcases...
testcases compile OK
[ add-longlong] PASS!
[ add] PASS!
[ 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 4-2 指令回归测试

4.4 IO 指令及功能的实现

对于一个冯诺依曼计算机，仅仅能够执行指令是不够的，我们还需要与外部进行交互。由于 x86 指令集使用单独的指令实现 IO 操作，因此我们首先需要实现 in/out 指令，由于 nemu 中模拟相关硬件的软件代码已在框架代码中给出，我们需要实现的仅为系统调用 `_io_read/_io_write` 时的相关行为，其相当于对 in/out 指令的一层封装。

此处我们需要实现的 IO 功能主要包括串口、时钟、键盘以及 VGA 显示器。串口端口在 nemu 中被绑定到了标准输出上，且相关功能已经实现，当我们实现了 in/out 指令后，即可使用串口功能来将字符串等数据输出到标准输出了。下面我们对余下三种 IO 进行分别说明。

4.4.1 时钟

时钟相关的 IO 中，我们需要实现的是一个获取系统启动到目前经过的时间的软件功能。首先，在时钟 IO 初始化时，我们使用 in 指令从 nemu 模拟的时钟硬件获取一个时间，将其作为 `boot_time` 保存；待系统通过 `_io_read` 读取时间时，我们再次使用 in 指令读取当前时间并将其与 `boot_time` 相减，得到的差值即为我们所需的系统启动到目前所经过的时间。

4.4.2 键盘

键盘 IO 需要实现的功能为读取来自键盘的按键信息，该信息包括两部分，被按的按键以及该按键被按下还是被松开。事实上，该信息我们可以直接使用 in 指令从相应的 IO 端口读出，按照文档及框架代码要求，对读出的信息进行解读和重新编码，返回给上层即可。

4.4.3 VGA 显示器

上述时钟与键盘均为只读设备，此处的 VGA 显示器则为可读可写设备，其读操作较为简单，即读取显示器的长宽信息，此信息我们可以直接使用 in 指令从相应硬件端口获取，解析后返回即可。其写操作则较为复杂，我们需要实现的主要是 `draw_rect` 函数，该函数用于在显示器上的一个矩形区域绘图，根据其传入的长宽等参数，我们需要计算各行需要赋值的像素点的位置，最终将制定内存区域的数据拷贝到预期的显

存区域，最后，将控制信号通过 out 指令写回到显示器硬件对应的端口中，待显示器刷新即可完成区域的显示。

4.5 运行结果

在实验过程中我们已经对各功能进行了基本的测试。下面，我们给出框架代码中的一些 app 在 nemu 上的运行结果。如图 4-3 为播放幻灯片测试结果，图 4-4 为打字游戏测试结果，图 4-5 为 microbench 运行结果，图 4-6 为任天堂模拟器运行游戏气球的运行结果。由上述 apps 的运行结果我们可以认为我们在 PA2 中实现的指令以及 IOE 相关的功能是正确的。

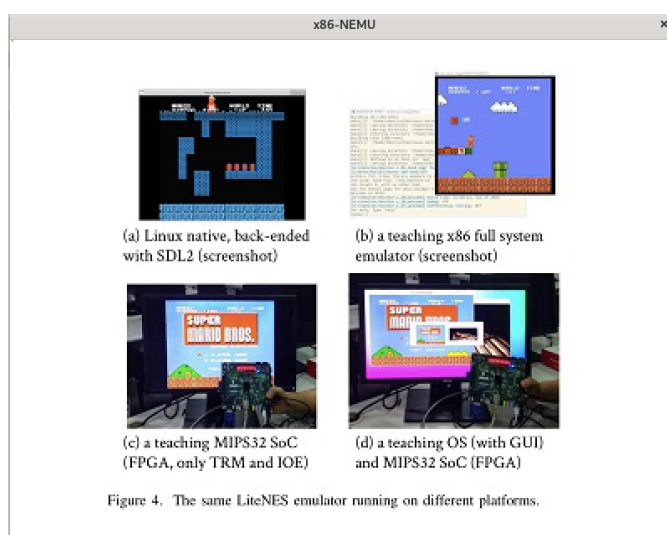


图 4-3 幻灯片测试(slider)



图 4-4 打字游戏测试(typing)

```
Welcome to x86-NEMU!
For help, type "help"
Empty mainargs. Use "ref" by default
===== Running MicroBench [input *ref*] =====
[qsort] Quick sort: * Passed.
min time: 1007 ms [507]
[queen] Queen placement: * Passed.
min time: 1230 ms [382]
[bf] Brainf**k interpreter: * Passed.
min time: 6435 ms [367]
[fib] Fibonacci number: * Passed.
min time: 14027 ms [201]
[sieve] Eratosthenes sieve: * Passed.
min time: 12070 ms [326]
[15pz] A* 15-puzzle search: * Passed.
min time: 2278 ms [196]
[dinic] Dinic's maxflow algorithm: * Passed.
min time: 2581 ms [421]
[lzip] Lzip compression: * Passed.
min time: 3129 ms [242]
[ssort] Suffix sort: * Passed.
min time: 1100 ms [409]
[md5] MD5 digest: * Passed.
min time: 13207 ms [130]
=====
MicroBench PASS      318 Marks
vs. 100000 Marks (i7-7700K @ 4.20GHz)
Total time: 66416 ms
nemu: HIT GOOD TRAP at pc = 0x00104288

[src/monitor/cpu-exec.c,33,monitor_statistic] total quest instructions = 1818608482
make[1]: 离开目录 /home/huao/Documents/code/ics2019/ics2019/nemu"
```

图 4-5 MicroBench 测试



图 4-6 litenes 测试

4.6 问题解答

最后我们对文档中的必答题进行解答。

1. nemu 中一条指令的执行流程在 4.1 节中已经给出了详细的描述，图 4-1 也给出了相应的流程图，此处不再赘述。

2. 经过尝试，我发现去掉 static 或 inline 修饰符都不会导致报错。

3. make 命令如何组织文件完成编译过程。在 makefile 中，首先我们根据输入设置了指令集 ISA，根据该变量，我们设置头文件搜索路径，包括目录 nemu/include 和 nemu/src/isa/x86/include，在 gcc 的编译选项中设置上述两个头文件路径。接下来，我们使用 find 等命令找到所有需要编译的源文件，首先找到 nemu/src 下除 isa 目录外其

华中科技大学课程设计报告

余所有位置的.c 文件，然后找到 `nemu/src/isa/x86` 目录下的全部.c 文件，这两部分加在一起即为待编译的源文件列表。最后，根据 `makefile` 中设置的编译器、编译选项等完成编译链接过程即可。

5 PA3 穿越时空的旅程: 批处理系统

5.1 上下文管理

这里我们通过描述一次自陷操作经过的全过程,对上下文管理的相关内容进行说明。首先,这里的自陷操作在 x86 中指的是执行 int 指令。以_yield 函数中执行的 int 0x81 指令为例,其在框架代码中经历的代码流程如下。

1. 文件 nexus-am/am/src/x86/nemu/cte.c 中, _yield 函数中执行指令 int 0x81。
2. 文件 nemu/src/isa/x86/exec/system.c 中, exec_int 函数为 nemu 对 int 指令的执行函数,其中调用了 raise_intr 函数,参数为 int 中断编号(此处为 0x81)以及当前的 PC 值。
3. 文件 nemu/src/isa/x86/intr.c 中, raise_intr 函数中读取中断描述符表 idt,根据传入的中断编号得到中断处理程序的入口地址(中断描述符表的初始化在 _cte_init 函数中完成)。根据要求,我们依次对寄存器 eflags 和 cs 进行压栈,对传入的 PC 值,即中断返回地址进行压栈,最后将程序转移到中断处理程序入口地址处继续执行。当中断编号为 0x81 时,在文件 nexus-am/am/src/x86/nemu/cte.c 中的 _cte_init 函数中我们可以看到中断处理程序为 __am_vectrap 函数,因此 raise_intr 最终的效果是将虚拟机内部运行的程序转移到了其中断服务程序处继续执行。经过上述 2 和 3 的操作, nemu 完成了 int 指令的执行。
4. 文件 nexus-am/am/src/x86/nemu/trap.S 中, __am_vectrap 函数包含如下操作,将整数 0x81 入栈,跳转到 __am_asm_trap 继续执行。该位置与 __am_vectrap 在同一文件中,其进行一系列压栈操作后,转移到函数 __am_irq_handle 处执行。
5. 文件 nexus-am/am/src/x86/nemu/cte.c 中,函数 __am_irq_handle 根据栈(上下文)中保存的中断号对事件进行打包,调用 user_handler 对事件进行处理。其中 user_handler 在 _cte_init 中进行了初始化,为 do_event 函数。
6. 文件 nanos-lite/src/irq.c 中, do_event 函数对传入的时间进行解析,做出相应的操作,对于 yield 操作,我们直接输出一段文本,表示程序运行至此即可。我们现在仅有一个上下文,因此不做上下文切换,此处直接返回 NULL 给 __am_irq_handle。
7. 接下来就是沿着上述调用链逐级返回的操作。 __am_irq_handle 得到 do_event

返回的 NULL 后,不做上下文切换,直接将传入的上下文返回给调用者 `__am_asm_trap`, 经过适当的出栈操作后,使用 `iret` 指令进行中断返回,回复现场,回到中断前程序的执行位置。至此,一次自陷操作全部完成。

在了解自陷操作的流程后,我们实际上就可以很轻易的将框架代码中缺少的部分加以补充实现。

5.2 用户程序、系统调用、文件系统

对于这一部分,我们首先来介绍系统调用。系统调用本质上与我们在 5.1 节中描述的 `yield` 自陷操作相同,就是一次中断的过程,只不过前面 `yield` 导致的中断我们几乎没有做任何事,而系统调用产生的中断就是要完成一些特定的工作。

系统调用发生的流程与 `yield` 类似,经过层层打包和转发,系统调用中断最终的处理程序落实到 `do_syscall` 函数中。我们在该函数中对系统调用号进行识别,然后做出相应的操作即可完成对系统调用的实现。

接下来我们来讨论文件系统。在 `linux` 中,我们采用了“一切皆文件”的思想,那么在课设中这个简易操作系统上,我们也做类似的设计,将串口、显示器等 IO 设备都抽象成文件形式,在操作系统层面统一使用文件方式进行访问。这种抽象实际上是对各种设备的一种封装,在文件 `nanos-lite/src/fs.c` 中,我们在数组 `file_table` 中为每种不同的文件指定对应的读写函数,这样,当我们调用文件系统中的读写操作时,最终将落实到这种特定设备文件的读写函数上,从而达成封装的目的。

当然,文件系统最终也是通过系统调用的方式加以实现,我们向系统中添加相应的系统调用,同时删除原有的设备相关的系统调用,即可完成这个简单的文件系统。

最后我们在文件系统的基础上加载用户程序。程序加载的关键就在于对其 `elf` 信息的解析。从程序加载的角度,我们需要关注的是程序文件的程序头(Program Header)。首先,我们读取文件的 `elf` 头,其位于文件开始的位置,从 `elf` 头中,我们可以知道该文件的程序头的位置及数量等信息。逐个读取其所有的程序头,若某个程序头带有需要加载的标识,我们就从中读出其对应的数据在文件中的位置以及其需要加载的内存的位置,完成程序的加载。遍历整个程序头表,将需要加载的数据拷贝到指定的内存区域,也就完成了程序的加载。最后,在文件的 `elf` 头中包含有该程序的入口地址,读出该数据,将其返回给操作系统,下一步操作系统将从该位置开始执行,这样程序便能够执行起来。

5.3 运行结果

这一部分中其实存在不少的测试程序，但总的来说，其目的在于验证各个部件的正确性，而最终的大程序“仙剑奇侠传”可以覆盖全面的这些测试点，因此这里我们仅展示“仙剑奇侠传”的运行结果。如图 5-1 所示，该程序可以正常启动并播放开场动画，如图 5-2 所示，游戏可以正常运行并播放对话。由此我们可以得知，PA3 的内容基本完成。

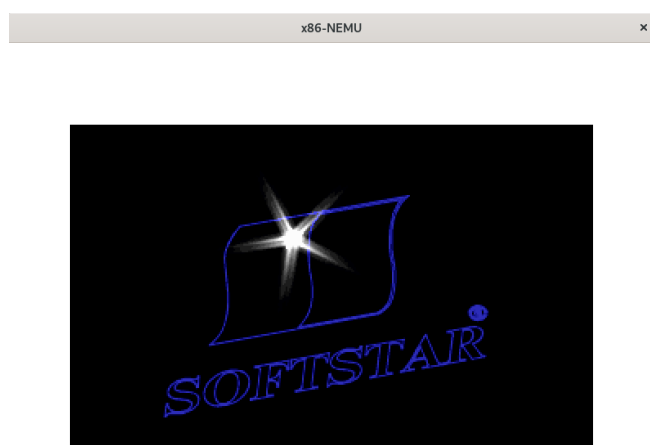


图 5-1 仙剑奇侠传开场动画



图 5-2 仙剑奇侠传游戏对话

5.4 问题解答

1. 上下文结构体。函数 `_am_irq_handle` 中，上下文结构体 `c` 是函数的参数，该函数是 `trap.S` 中通过 `call` 指令调用的，根据 x86 栈帧的结构，参数 `c` 是最后压入栈中

的那个值，即压栈前的`%esp`寄存器的值，因此指针 `c` 指向再前面一条指令压入栈中的常数 `0`。指针的本质就是一个内存地址，因此结构体 `c` 的内容就是那一段栈内存空间中的数据，也就是我们通过不断压栈保存的数据。根据指令集中对 `int` 指令的实现以及 `trap.S` 中压栈的流程，我们可以知道，上下文结构体中的内容依次是一个暂时作用位置的常数 `0`、八个通用寄存器按照 `pusha` 的顺序反向排列、中断号、中断返回地址、`cs` 寄存器以及 `eflags` 寄存器的值。

2. 自陷操作的全过程。该过程在 5.1 节中已经给出了详细的描述，包括整个函数调用链，并给出了每个函数所在的文件。

3. `hello` 程序运行的全过程。在课设中实现的系统中也好、在真机上的 `linux` 上也好，其流程大致类似，因此下面我们不特别强调 `nemu` 中的虚拟设备。

编译完成后，`hello` 程序在磁盘上。当用户运行该程序时，首先操作系统有一个程序加载器，它将 `hello` 程序从磁盘读出，根据其 `elf` 头信息加载到指定的内存空间，加载完成后，操作系统从其 `elf` 信息中获取到程序入口地址，通过上下文切换从该入口地址处继续执行。至此，`hello` 程序便获取到 CPU 的控制权开始执行指令。

至于字符串在终端的显示，首先程序调用 `printf` 等库函数，这些函数调用相应的系统调用，系统调用通过调用外设的驱动程序最终将内容在外设中表现出来。

6 PA4 虚实交错的魔法: 分时多任务

6.1 多道程序

在前面的实验中,我们将程序加载到特定的内存地址,倘若要加载多个应用程序,那么所有的程序将共用一个内存空间,这样,一方面程序的加载、程序能够访问的内存空间控制都较为复杂,且会存在很大的安全隐患。因此,我们通过虚拟内存的方式来实现多道程序,每个程序都有其独立的内存空间,通过内存分页机制来访问内存,这样,每个程序只能看到其自己的内存空间,既解决了程序加载的麻烦,也对不同程序进行了隔离。

首先,我们需要实现内存分页机制。这里我们主要需要完成两部分内容,一是将虚地址映射到实地址,一是对虚地址的访问。虚实地址映射本质上就是页表,因此对于前者,我们要做的工作就是根据两个地址填写页表,后者则是读取页表将虚地址转换为实地址,从而进行访存。

关于虚拟内存,我们在操作系统课程中已经有了详细的讲解,下面我们将以将虚地址转换为实地址的过程为例,对实验中使用的二级页表的分页机制进行描述。

虚地址 `vaddr` 是一个 32 位地址,我们将其高 10 位命名为 `dir`、低 12 位命名为 `offset`、余下 10 位命名为 `page`。那么获取其对应的实地址的方法如下。首先寄存器 `CR3` 中保存了当前程序的页目录的基地址,找到其中的第 `dir` 项,其内容为一个页表的基地址,找到该页表的第 `page` 项,其内容为一个内存页的基地址,将该基地址与 `offset` 组合在一起,即为虚地址 `vaddr` 对应的实地址。当然,页目录/页表中的内容还包括一些控制位,上面没有详细说明,但不影响 `page walk` 的大致流程。

在 `nemu` 中添加必须的寄存器,增加相应的指令,完成 `mmu` 的功能即访问虚地址,在 `am` 中完成虚实地址映射功能,在 `nanos-lite` 中实现内存页分配功能,这样我们就基本实现了内存分页机制。

实现分页机制后,我们需要对 `PA3` 中实现的程序加载功能进行修改。首先,在加载程序前后,我们需要为其创建地址空间、上下文等内容,另外,现在程序需要加载到对应的虚地址的位置,另外我们需要以页为单位进行加载,加载前需要申请相应的内存空间并将其与虚地址进行映射。

至此，我们就可以在支持分页机制的操作系统上运行程序了。

6.2 分时多任务

最后我们来实现进程调度相关的内容。最简单的调度方式莫过于当一个程序不再使用 CPU 时就将 CPU 使用权交给另一个程序，另一个程序继续运行，此处，不再使用 CPU 包括两种情况，一种是暂时不再使用，例如进行 IO 操作等，另一种是程序结束。这样，我们可以前面已经实现的 `yield` 操作来进行实现。在 `yield` 产生的中断中进行进程调度完成上下文切换，待中断返回后将从新的程序处开始运行。我们在 IO 操作前调用 `yield` 函数，即可实现这种简单的调度方式。

当然，上述调度方式并不是很好，倘若一个程序需要进行长时间的计算而不与外设交互，那么它将长时间占据 CPU，其他程序只能陷入长时间的等待。下面我们实现一种基于时间片轮转的进程调度。首先，我们在 `nemu` 中实现一个每间隔固定时间就触发一次的时钟中断，在每条指令执行完成后，检查是否存在时钟中断，若存在，则清除中断信号并进行进程调度。这样，系统中运行的所有进程就可以轮流使用 CPU。

最后，我们赋予三个按键 `F1/F2/F3` 一些特殊的功能，让三个用户程序同时运行，使用这三个按键就当前屏幕显示的程序进行切换。其运行方式及效果我们在 6.3 节中进行详细的描述。

6.3 运行结果

在 `nanos-lite` 目录下，执行 `make ARCH=x86-nemu run` 命令启动机器及操作系统。由于此处需要展示的功能是在按键过程中不断出现的变化，因此不便通过截图展示，因此此处我们仅通过文字对其进行描述。

程序启动以后，我们会看到一个程序菜单，此时我们按下按键 3，启动仙剑奇侠传，可以看到游戏的启动动画，此时我们按下按键 F2，可以看到显示器显示内容变为程序菜单，我们再次按下按键 3，可以看到游戏新的游戏启动画面，经过一段时间后，我们按下按键 F1，可以看到显示器画面切换回上次按下 F2 之前的状态，通过 `F1/F2` 可以在两个画面中间不断切换。类似的，我们可以用上 `F3`，运行第三个应用程序。

6.4 问题解答

分时多任务的具体流程。以时间片轮转的调度方式为例，假设当前正在前台运行

华中科技大学课程设计报告

的程序是仙剑奇侠传，我们不妨称其为 `pal`，则等待运行的进程为 `hello`。`pal` 不断执行指令，直到某条指令执行完成后，`nemu` 检测到一个时钟中断，并通过一系列操作进入中断服务程序开始运行，此次中断服务程序中，我们调用 `yield` 函数触发另一个中断，在此中断中，调用 `schedule` 函数进行进程调度，根据调度策略，其将返回 `hello` 进程的上下文，待到中断结束，`iret` 指令执行时，上下文被替换成 `hello` 进程，机器从 `hello` 进程上次停下的位置继续执行。时钟中断每间隔一段时间将到来一次，因此上述操作每间隔一段时间将发生一次，进行进程切换。

7 总结与心得

7.1 课设总结

本次课程设计中，我们完成了如下工作。

1. 实现了一个简单调试器，可以执行单步/多步运行、断点、表达式求值等功能。
2. 完成了 x86 虚拟机 `nemu` 的指令实现，并为其添加了 IO、中断、虚存等支持。
3. 完成了简易操作系统 `nanos-lite`，支持简单的进程管理、内存管理、文件系统等功

7.2 课设心得

本次课程设计的过程可谓是一波三折，最初为了方便使用了虚拟机，后来由于性能以及虚拟机共享文件的权限等问题改用真机双系统进行实验，其间遇到了一些匪夷所思的环境问题，又由于课设截止时间较晚，虽然很早便开始了课设的内容，但是中途做做停停，因此前后花费了很长的时间，而报告的编写则拖到了更晚的时候。不过，值得欣慰的是，最终我们实现了 PA 的大部分功能，能够在虚拟机 `nemu` 上运行起支持分时多任务的简易操作系统，并在其上同时运行多个类似于仙剑奇侠传这种大规模的应用程序。

这次 PA 串联了我们在大学前三年里学习的很多内容，从 PA1 中基本的程序设计和算法，到 PA2 中涉及到的汇编语言、组成原理等，再到 PA3 和 PA4 中的操作系统。通过完成 PA 的内容，我回顾了很多以前学习过的知识，对其中的一些内容也有了新的认识，但我认为这门课程还是存在很多需要改进的地方。

首先是关于实验内容，在 PA3 和 PA4 中，除了中断和 `mmu` 与硬件 `nemu` 相关外，其余的部分均为操作系统的内容，但是由于此实验原本是设计给尚未学习过操作系统课程的大二同学做的，因此其中模糊、简化了很多概念，这导致这部分内容对我们大四的同学而言有些鸡肋，食之无味弃之可惜。我曾在操作系统课程设计期间自行完成过 MIT 6.828 操作系统实验的内容，在有其基础的情况下进行 PA3/PA4 的实验内容，我觉得内容确实较为简单，而其中故意对操作系统中的概念进行的模糊和简化，例如固定数量的进程、没有目录且文件数目固定的文件系统等，给人一种非常别扭的感觉。

华中科技大学课程设计报告

我知道我们的操作系统课程设计较为简单，主要还是对 Linux 的熟悉和使用，但是倘若对操作系统课设进行一定的改革，实现一个类似于 JOS 的简易操作系统，然后在本课程中设计虚拟机并与之相结合，或许能有更好的学习效果。

另外，对于实验文档，从课程群的讨论可以看出，很多同学对其颇有微词。我觉得南京大学提供的这份 PA 的文档有些地方确实有些主次不分，对于有些实验中的重点难点，其故意一笔带过，美其名曰让我们自行思考，却经常花费大量篇幅提一些无关紧要的内容。以 x86 版本的 PA2 为例，这部分实验的主要内容就是对指令集的实现，x86 作为变长指令、寻址方式众多的一种复杂指令集，其译码过程可谓相当复杂。在实验框架代码中提供了很多的已经实现好的函数可供调用，然后对于这些函数，实验文档却几乎只字不提其作用，代码中虽有注释，却也是含糊不清，其各种缩写若不加解释让人不知所云。我觉得，我们可以在南京大学的 PA 文档外，另外编写一份简单的补充文档，对实验中的重难点进行一定的讲解，这对我们理解和完成实验内容都会有很好的帮助。

最后，我希望这门课程设计可以增加一些自动测试和评分的系统，就像组成原理实验与课设中使用的 `educoder` 自动测试那样。对于 PA1 的内容，主要是程序设计和算法的内容，使用大量的样例进行测试即可；对于 PA2 的内容，由于指令执行的结果具有确定性，因此比较每条指令执行后的处理器状态就可以验证对错。这两部分，虽然我们在实验中也实现了 `qemu-diff` 这样的功能，但是倘若老师能够提供一套评测系统，我们在实验中也可以更加准确的对实现结果正确与否进行认定。至于 PA3 和 PA4 这些与操作系统相关的内容，参考课程 6.828 的设计，同样可以提供一套测试评分功能。我认为，对于 PA 这种以功能复现为目的的实验，自动测评对我们的实验可以起到很大的帮助，它能够让我们准确的判断和掌握实验进度，不至于到了实验后期才发现前面的 bug，导致大量的更改。

参考文献

- [1] DAVID A. PATTERSON(美). 计算机组成与设计硬件/软件接口(原书第 4 版). 北京: 机械工业出版社.
- [2] David Money Harris(美). 数字设计和计算机体系结构(第二版). 机械工业出版社
- [3] 秦磊华, 吴非, 莫正坤. 计算机组成原理. 北京: 清华大学出版社, 2011 年.
- [4] 谭志虎, 秦磊华, 胡迪青. 计算机组成原理实践教程. 北京: 清华大学出版社, 2018.
- [5] 袁春风编著. 计算机组成与系统结构. 北京: 清华大学出版社, 2011 年.
- [6] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008 年.