***2017***



**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： | ACM1501 |
| 学 号： | U201514566 |
| 姓 名： | 董超 |
| 电 话： | 15927382865 |
| 邮 件： | [948914729@qq.com](mailto:948914729@qq.com) |
| 完成日期： | 2019-01-09 周三晚 |

目 录

[1 课程设计概述 2](#_Toc497213218)

[1.1 课设背景 2](#_Toc497213219)

[1.2 设计任务 2](#_Toc497213220)

[2 PA1 简易调试器 3](#_Toc497213221)

[2.1 功能实现的要点 3](#_Toc497213222)

[2.2 必答题 4](#_Toc497213223)

[2.3 主要故障与调试 5](#_Toc497213224)

[3 PA2 冯诺依曼计算机系统 7](#_Toc497213225)

[3.1 功能实现的要点 7](#_Toc497213226)

[3.2 必答题 9](#_Toc497213227)

[3.3 主要故障与调试 11](#_Toc497213228)

[4 PA3 异常控制流 13](#_Toc497213229)

[4.1 功能实现的要点 13](#_Toc497213230)

[4.2 主要故障与调试 15](#_Toc497213231)

[5 设计总结与心得 17](#_Toc497213232)

[5.1 课设总结 17](#_Toc497213233)

[5.2 课设心得 17](#_Toc497213234)

|  |
| --- |
| 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:** |

# **课程设计概述**

## 课设背景

NEMU受到了QEMU的启发, 结合了GDB调试器的特性, 去除了大量与课程内容差异较大的部分，该课程的学习将帮助大家了解模拟器系统开发的基本思路，熟悉掌握阅读大型软件项目代码框架的技巧和方法，通过实现X86模拟器，将对X86 CPU软硬件系统有着更为清晰的理解。

NEMU是一个X86模拟器，是一个普通的用户态应用程序, 在GNU/Linux操作系统中运行，但同时又是一个特殊的程序, 可虚拟出一个计算机系统, 其它程序可以在其中模拟执行，通过模拟器的设计让学生理解掌握下图中X86硬件基本构成以及软硬协同的机制。



图1 NEMU模拟器计算机框架示意图

学生将首先设计实现一款类似GDB的调试工具，然后利用程序模拟实现X86 CPU，寄存器组，并利用程序实现X86机器指令取指、指令译码，指令执行的动作，程序解释执行过程中，还必须实现虚拟存储系统的管理机制，完成虚拟地址到线性地址到物理地址的转换，为加速程序执行还必须完成CPU Cache机制的模拟，另外为支持程序的输入输出，还必须利用程序实现X86中断异常处理以及I/O处理机制，最终见证奇迹的剑奇侠传移植工作将向你展示了ISA如何把软件硬件联系起来, 从而支持一个游戏的运行，整个模拟器的设计实现贯穿计算机组成原理的方方面面，也涉及操作系统、编译原理相关内容，有利于进一步加深对计算机分层系统栈的理解，梳理大学3年所学的全部理论知识，大大提升学生计算机系统能力。

## 设计任务

PA项目包括一个准备实验(配置实验环境)以及4部分连贯的实验内容，具体时间和任务分布如下，该项目提供了总体框架代码，学生只需要完成框架代码中的部分代码，故总体代码量只有2000多行，但阅读代码框架是一项较大的挑战。

表1.1 NEMU模拟器任务量统计表



# **PA1 简易调试器**

## 功能实现的要点

(1) 正确地实现用于模拟寄存器的结构体CPU\_state (结构体的定义在nemu/include/cpu/reg.h中) 。

在本节任务中，我们需要实现正常的寄存器结构体。原框架给出的代码如下：

|  |
| --- |
| 1 typedef struct {  2 struct {  3 uint32\_t \_32;  4 uint16\_t \_16;  5 uint8\_t \_8[2];  6 } gpr[8];  7  8 /\* Do NOT change the order of the GPRs' definitions. \*/  9  10 /\* In NEMU, rtlreg\_t is exactly uint32\_t. This makes RTL instructions  11 \* in PA2 able to directly access these registers.  12 \*/  13 rtlreg\_t eax, ecx, edx, ebx, esp, ebp, esi, edi;  14  15 vaddr\_t eip;  16  17 } CPU\_state |

CPU包含8个基本寄存器和1个eip寄存器，每个寄存器总大小为4字节，并且可以按照2字节，1字节访问。显然，这里应该使用匿名联合。

|  |
| --- |
| 1 typedef struct {  2 union{  3 union {  4 uint32\_t \_32;  5 uint16\_t \_16;  6 uint8\_t \_8[2];  7 } gpr[8];  8  9 /\* Do NOT change the order of the GPRs' definitions. \*/  10  11 /\* In NEMU, rtlreg\_t is exactly uint32\_t. This makes RTL instructions  12 \* in PA2 able to directly access these registers.  13 \*/  14 struct {  15 rtlreg\_t eax, ecx, edx, ebx, esp, ebp, esi, edi;  16 };  17 };  18 vaddr\_t eip;  19 } CPU\_state; |

在上述代码中，gpr的类型为匿名联合，作用是使同一寄存器变量指向的内存地址相同。而较外层的联合，作用是给gpr数组中元素重命名为eax，ecx…等寄存器名称。

(2)完成简易调试器

简易调试器是NEMU中一项非常重要的基础设施. NEMU是一个用来执行其它客户程序的程序, 这意味着, NEMU可以随时了解客户程序执行的所有信息. 然而这些信息对外面的调试器(例如GDB)来说, 是不容易获取的。

例如在通过GDB调试NEMU的时候, 很难在NEMU中运行的客户程序中设置断点, 但对于NEMU来说, 这是一件不太困难的事情。为了提高调试的效率, 同时也作为熟悉框架代码的练习, 我们在monitor中实现一个具有如下功能的简易调试器。

表1.2 NEMU调试器命令格式和功能对照表

| **命令** | **格式** | **使用举例** | **说明** |
| --- | --- | --- | --- |
| 帮助(1) | help | help | 打印命令的帮助信息 |
| 继续运行(1) | c | c | 继续运行被暂停的程序 |
| 退出(1) | q | q | 退出NEMU |
| 单步执行 | si [N] | si 10 | 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 | info r info w | 打印寄存器状态 打印监视点信息 |
| 表达式求值 | p EXPR | p $eax + 1 | 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](https://nju-ics.gitbooks.io/ics2018-programming-assignment/content/1.6.html)小节 |
| 扫描内存(2) | x N EXPR | x 10 $esp | 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 | w \*0x2000 | 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 | d 2 | 删除序号为N的监视点 |

在上述功能中，监视点的设置和删除即操作链表，其他功能都比较容易实现，只有表达式求值比较困难。

为了解决表达式求值的问题，我们需要完成这样几个任务：词法分析，递归求值，寻找主运算符。

词法分析的任务比较简单，我们使用正则表达式完成词法识别，并且保存为不同类型的结点(使用token结构记录)。

递归求值的主要思路：

a.如果表达式中只有一个token，直接返回这个token的，否则转b；

b.如果表达式被括号包围，去掉括号重新求值，否则转c；

c.寻找主操作符，分别计算主操作符左右两边的值，在计算。

上述过程我们设计到主操作符的使用，主操作符的定义非常简单，即最后进行的运算符号。我们可以通过下述方式寻找主操作符：

a.非运算符的token不是主运算符；

b.出现在一对括号中的token不是主运算符。注意到这里不会出现有括号包围整个表达式的情况。

c.主运算符的优先级在表达式中是最低的。这是因为主运算符是最后一步才进行的运算符。当有多个运算符的优先级都是最低时，根据结合性，最后被结合的运算符才是主运算符. 一个例子是1 + 2 + 3, 它的主运算符应该是右边的+.

上述过程大致描述了计算表达式的思路，但接下来有一个重要的问题，如何判断是否被括号包围以及如何寻找主操作符。

我们给每个表达式构造一个辅助数组给定一个算式（1+2）\*（3\*4），括号匹配的方式如下图所示。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ( | 1 | + | 2 | ) | \* | ( | 3 | \* | 4 | ) |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

图1.2 括号匹配辅助数组示意图

在上图中，第一行是给定的算式，第二行是构造的辅助数组。辅助数组与算式等长，位置一一对应。当算式中对应的符号为“(”时，数组相应内容为前一位加一；为“)”时，内容为前一位减一。假设第零位是0。

我们可以从辅助数组中看出括号匹配情况。当且仅当辅助数组所有内容非负，并且最后一位为0时，括号匹配正常。并且可以寻找主操作符，在对应辅助数组值为0的运算符中优先级最低的符号（相同优先级时选择在最右侧的符号），即为主操作符。

通过上述过程，我们可以完成不含单目运算的表达式的计算。如果表达式需要支持单目运算（如负号，取址等），我们可以通过判断符号前方的token类型来判断运算符号的类型。

## 必答题

a.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：500 \* 90% \* 30 \* 20 = 270,000 = 75h

75 \* (30 - 10) / 30 = 50h

b.查阅i386手册 理解了科学查阅手册的方法之后, 请你尝试在i386手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:

EFLAGS寄存器中的CF位是什么意思?

ModR/M字节是什么?

mov指令的具体格式是怎么样的?

答：CF用于判断进位标记（page419）

ModR/M字节用于取址。0、1、2字节为R/M部分，3、4、5为REG/OPCODE部分，6、7为MOD部分。MOD和R/M部分用于确定目的操作数的地址，REG/OPCODE为源操作数或操作指令。（page242-245）

mov指令的包含多种具体格式，具体可参考Inter 80386手册P345页。

c.shell命令 完成PA1的内容之后, nemu/目录下的所有.c和.h和文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在PA1中编写了多少行代码? (Hint: 目前pa1分支中记录的正好是做PA1之前的状态, 思考一下应该如何回到统计工程的代码行数, 例如敲入make count就会自动运行统计代码行数的命令. 再来个难一点的, 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：统计.c文件：find . -name "\*.c" |xargs cat|wc –l



统计.h文件：find . -name "\*.h" |xargs cat|wc –l



忽略空行统计：

find . -name "\*.c" |xargs cat|grep -v ^$|wc –l

find . -name "\*.h" |xargs cat|grep -v ^$|wc -l

d.使用man 打开工程目录下的Makefile文件, 你会在CFLAGS变量中看到gcc的一些编译选项. 请解释gcc中的-Wall和-Werror有什么作用? 为什么要使用-Wall和-Werror?

答：

-Wall 打开gcc的所有警告;

-Werror，它要求gcc将所有的警告当成错误进行处理。

使用Wall和Werror可以提高我们程序的正确率，避免后期由于前面的错误而拖慢进度。

## 主要故障与调试

a.union匿名联合的使用

在寄存器设计的过程中，我需要是用匿名联合。刚开始设计时我使用了下列代码。但发现结果不正确，因而仔细查看，发现问题。所有寄存器所在的匿名联合包含的成员为：gpr，eax，ecx，edx，ebx，esp，ebp，esi，edi，这九个元素共享一块内存，显然这是不符合要求的。

|  |
| --- |
| 1 typedef struct {  2 union{  3 union {  4 uint32\_t \_32;  5 uint16\_t \_16;  6 uint8\_t \_8[2];  7 } gpr[8];  8  9 /\* Do NOT change the order of the GPRs' definitions. \*/  10  11 /\* In NEMU, rtlreg\_t is exactly uint32\_t. This makes RTL instructions  12 \* in PA2 able to directly access these registers.  13 \*/  14  15 rtlreg\_t eax, ecx, edx, ebx, esp, ebp, esi, edi;  16  17 };  18 vaddr\_t eip;  19 } CPU\_state; |

解决方法：将eax，ecx，edx，ebx，esp，ebp，esi，edi，八个寄存器再放在同一结构体中，该结构体与gpr位于同一匿名联合，共享一片内存。

# **PA2 冯诺依曼计算机系统**

## 功能实现的要点

1. 实现新指令

在NEMU模块中，新指令的执行流程是比较复杂的，主要分为译码和执行两个步骤。首先将当前的%eip保存到全局译码信息decoding的成员seq\_eip中，然后将其地址被作为参数送进exec\_real()函数中。seq代表顺序的意思，当代码从exec\_real()返回时，decoding。seq\_eip将会指向下一条指令的地址。exec\_real()函数通过宏make\_EHelper来定义：

#define make\_EHelper(name) void concat(exec\_，name) (vaddr\_t \*eip)

其含义是"定义一个执行阶段相关的helper函数"，这些函数都带有一个参数eip。NEMU通过不同的helper函数来模拟不同的步骤。

在exec\_real()中：

* 首先通过instr\_fetch()函数(在nemu/include/cpu/exec.h中定义)进行取指，得到指令的第一个字节，将其解释成opcode并记录在全局译码信息decoding中。
* 根据opcode查阅译码查找表，得到操作数的宽度信息，并通过调用set\_width()函数将其记录在全局译码信息decoding中。
* 调用idex()对指令进行进一步的译码和执行

idex()函数会调用译码查找表中的相应的译码函数进行操作数的译码。译码函数统一通过宏make\_DHelper来定义(在nemu/src/cpu/decode/decode.c中)：

#define make\_DHelper(name) void concat(decode\_，name) (vaddr\_t \*eip)

它们主要以i386手册附录A中的操作数表示记号来命名，例如I2r表示将立即数移入寄存器，其中I表示立即数，2表示英文to，r表示通用寄存器，更多的记号请参考i386手册。译码函数会把指令中的操作数信息分别记录在全局译码信息decoding中。

这些译码函数会进一步分解成各种不同操作数的译码的组合，以实现操作数译码的解耦。操作数译码函数统一通过宏make\_DopHelper来定义 (在nemu/src/cpu/decode/decode.c中，decode\_op\_rm()除外)：

#define make\_DopHelper(name) void concat(decode\_op\_，name) (vaddr\_t \*eip，Operand \*op，bool load\_val)

它们主要以i386手册附录A中的操作数表示记号来命名。操作数译码函数会把操作数的信息记录在结构体op中，如果操作数在指令中，就会通过instr\_fetch()将它们从eip所指向的内存位置取出。为了使操作数译码函数更易于复用，函数中的load\_val参数会控制 是否需要将该操作数读出到全局译码信息decoding供后续使用。例如如果一个内存操作数是源操作数，就需要将这个操作数从内存中读出来供后续执行阶段来使用; 如果它仅仅是一个目的操作数，就不需要从内存读出它的值了，因为执行这条指令并不需要这个值，而是将新数据写入相应的内存位置。

idex()函数中的译码过程结束之后，会调用译码查找表中的相应的执行函数来进行真正的执行操作。执行函数统一通过宏make\_EHelper来定义，它们的名字是指令操作本身。执行函数通过RTL来描述指令真正的执行功能。其中operand\_write()函数(在nemu/src/cpu/decode/decode.c中定义) 会根据第一个参数中记录的类型的不同进行相应的写操作，包括写寄存器和写内存。

从idex()返回后，exec\_real()也会返回到exec\_wrapper()中，最后会通过update\_eip()对%eip进行更新。

事实上，框架已经为我们做好了绝大部分工作，为了实现一条新指令，只需：

1. 在opcode\_table中填写正确的译码函数, 执行函数以及操作数宽度
2. 用RTL实现正确的执行函数

这里举例指令push的实现步骤。

a) 在nemu/include/cpu/rtl.h中实现rtl\_push函数。rtl\_push函数用于使cpu.esp自减四，再将某个寄存器中的值放入cpu.esp指向的内存中，

|  |
| --- |
| static inline void rtl\_push(const rtlreg\_t\* src1) {  // esp <- esp - 4  // M[esp] <- src1  cpu.esp -= 4;  vaddr\_write(cpu.esp, \*src1, 4);  } |

b) 编辑nemu/src/cpu/exec/exec.c中的指令表，再其中添加对应机器码的指令名称和译码规则。

|  |
| --- |
| /\* 0x50 \*/ EX(push), EX(push), EX(push), EX(push),  /\* 0x54 \*/ EX(push), EX(push), EX(push), EX(push),  …  /\* 0x68 \*/ IDEX(I, push), IDEX(I\_E2G, imul3), IDEXW(I, push, 1), IDEXW(I\_E2G, imul3, 1), |

c) 实现push指令的执行函数。

|  |
| --- |
| make\_EHelper(push) {  if(decoding.opcode >= 0x50 && decoding.opcode <= 0x57){  uint32\_t cc = decoding.opcode & 0x7;  id\_dest->val = id\_dest->width == 2 ? cpu.gpr[cc].\_16 : cpu.gpr[cc].\_32;  char reg\_name[10] = "%";  strcat(reg\_name, id\_dest->width == 2 ? regsw[cc] : regsl[cc]);  strcpy(id\_dest->str, reg\_name);  }  rtl\_sext(&id\_dest->val, &id\_dest->val, id\_dest->width);  rtl\_push(&id\_dest->val);  print\_asm\_template1(push);  } |

d) 将push执行函数添加到头文件nemu/src/cpu/exec/all-instr.h中。

|  |
| --- |
| make\_EHelper(push); |

1. 实现常用的库函数

我们已经在TRM上运行了不少简单的程序了，但如果想在TRM上编写一些稍微复杂的程序，我们就会发现有点不方便，目前TRM这个最简单的运行时环境只提供了堆区和\_halt()，但我们平时经常使用的像memcpy()这样的库函数却没有提供。 既然没有提供，那就让我们来实现一下吧。

既然叫得起库函数， 那说明很多程序都可以用到它们，所以我们可以像AM那样，把它们组织成一个库。 然而和AM不同的是，这些库函数的具体实现可以是和机器无关的: 与\_halt()不同，在NEMU上，或者在你将来用verilog实现的CPU上，甚至是其它的机器，memcpy()都可以通过相同的方式来实现。所以，如果在AM中来实现这些常用的库函数，就会引入不必要的重复代码。

一种好的做法是，把运行时环境分成两部分: 一部分是机器相关的运行时环境，也就是我们之前介绍的AM; 另一部分是机器无关的运行时环境，类似memcpy()这种常用的函数应该归入这部分。所以nexus-am/libs用于收录机器无关的库函数。

在PA中，我们只要关注nexus-am/libs/klib就可以了。klib是kernel library的意思，用于提供一些兼容libc的基础功能。框架代码在nexus-am/libs/klib/src/string.c和nexus-am/libs/klib/src/stdio.c 中列出了将来可能会用到的库函数，但并没有提供相应的实现。

再所有库函数中，最复杂的是类似printf的格式化输入/输出函数，因为输入的参数数量不确定。我们先实现vsprintf功能，vsprintf 是sprintf 的一个变形，它只有三个参数。vsprintf 用于执行有不定数量参数的函数，类似printf 格式。vsprintf的前两个参数与sprintf相同：一个用于保存结果的字符串缓冲区和一个格式化字符串。第三个参数是指向格式化参数队列的指针。实际上，该指针指向在堆栈中供函数调用的变量。va\_list、va\_start和va\_end宏（在STDARG.H中定义）帮助我们处理堆栈指针。

vsprintf的不定参数的问题通过上述方式得到解决。接下来我们要思考如何解决格式化的问题。这里的想法是通过分析源字符串，遇到‘%’符号时进入判断流程，并将格式化字符替换成判断输出结果。

在我的格式化处理函数中，只能分辨下述格式化字符(用正则表达式表示)：(0)[1-9]\*[d,c,x,s]。

其他格式化输出函数均通过调用vsprintf实现。

(3) 输入输出设备

CPU通过in和out指令访问外设，这里将访问外部设备的方式分成分成两类。

端口I/O：

一种I/O编址方式是端口映射I/O(port-mapped I/O)，CPU使用专门的I/O指令对设备进行访问，并把设备的地址称作端口号。有了端口号以后，在I/O指令中给出端口号，就知道要访问哪一个设备寄存器了。市场上的计算机绝大多数都是IBM PC兼容机，IBM PC兼容机对常见设备端口号的分配有专门的规定。

x86提供了in和out指令用于访问设备，其中in指令用于将设备寄存器中的数据传输到CPU寄存器中，out指令用于将CPU寄存器中的数据传送到设备寄存器中。一个例子是nexus-am/am/arch/x86-nemu/src/trm.c中\_putc()的代码，代码使用out指令给串口发送命令字。例如

movl $0x41，%al

movl $0x3f8，%edx

outb %al，(%dx)

上述代码把数据0x41传送到0x3f8号端口所对应的设备寄存器中。CPU执行上述代码后，会将0x41这个数据传送到串口的一个寄存器中，串口接收之后，发现是要输出一个字符A；但对CPU来说，它并不关心设备会怎么处理0x41这个数据，只会老老实实地把0x41传送到0x3f8号端口。

內存映射I/O

端口映射I/O把端口号作为I/O指令的一部分，这种方法很简单，但同时也是它最大的缺点。指令集为了兼容已经开发的程序，是只能添加但不能修改的。这意味着，端口映射I/O所能访问的I/O地址空间的大小，在设计I/O指令的那一刻就已经决定下来了。所谓I/O地址空间，其实就是所有能访问的设备的地址的集合。随着设备越来越多，功能也越来越复杂，I/O地址空间有限的端口映射I/O已经逐渐不能满足需求了。有的设备需要让CPU访问一段较大的连续存储空间，如VGA的显存，24色加上Alpha通道的1024x768分辨率的显存就需要3MB的编址范围。于是内存映射I/O(memory-mapped I/O)应运而生。

内存映射I/O这种编址方式非常巧妙，它是通过不同的物理内存地址给设备编址的。这种编址方式将一部分物理内存"重定向"到I/O地址空间中，CPU尝试访问这部分物理内存的时候，实际上最终是访问了相应的I/O设备，CPU却浑然不知。这样以后，CPU就可以通过普通的访存指令来访问设备。这也是内存映射I/O得天独厚的好处：物理内存的地址空间和CPU的位宽都会不断增长，内存映射I/O从来不需要担心I/O地址空间耗尽的问题。从原理上来说，内存映射I/O唯一的缺点就是，CPU无法通过正常渠道直接访问那些被映射到I/O地址空间的物理内存了。但随着计算机的发展，内存映射I/O的唯一缺点已经越来越不明显了：现代计算机都已经是64位计算机，物理地址线都有48根，这意味着物理地址空间有256TB这么大，从里面划出3MB的地址空间给显存，根本就是不痛不痒。正因为如此，内存映射I/O成为了现代计算机主流的I/O编址方式：RISC架构只提供内存映射I/O的编址方式，而PCI-e，网卡，x86的APIC等主流设备，都支持通过内存映射I/O来访问.

内存映射I/O的一个例子是NEMU中的物理地址区间[0x40000，0x80000)。这段物理地址区间被映射到VGA内部的显存，读写这段物理地址区间就相当于对读写VGA显存的数据。例如

memset((void \*)0x40000，0，SCR\_SIZE);

会将显存中一个屏幕大小的数据清零，即往整个屏幕写入黑色像素，作用相当于清屏。可以看到，内存映射I/O的编程模型和普通的编程完全一样：程序员可以直接把I/O设备当做内存来访问。

## 必答题

a.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：去掉static关键词后，发现发生错误：multiple definition of `interpret\_rtl\_li'。static关键词修饰函数，主要用于对其他源码文件隐藏本函数，起到防止命名冲突的问题。这里去掉static关键词后，函数名称产生冲突，因而编译错误。

inline关键词去掉后并没有发生错误。inline关键词对于程序的结果没有任何影响，只是为了避免频繁的出入栈操作。

b.编译与链接

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?

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.

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。)

答：在common.h添加变量dummy后，NEMU包含1个dummy变量实体。通过gdb的print指令打印dummy变量地址确定。

在debug.h中再次添加后，发现NEMU包含1个dummy变量实体。发现两次定义dummy实体，均指向同一内存区域，这是因为static关键词的特性。当第一次声明该变量时，会分配静态空间，而第二次执行声明操作时直接跳过。

当初始化后，发生报错：redefinition of ‘dummy’。这是因为含有初始化语句后，变量dummy出现变量重复定义问题。而如果去除初始化语句，则编译器会认为第二次为变量申明。

c.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。(这个问题包括两个方面:Makefile的工作方式和编译链接的过程。)

答：Makefile是用于编译连接运行程序的一个脚本，里面的内容可以用较为繁琐的命令行操作代替，因而Makefile是用于批处理的脚本文件。

在nemu/下的Makefile主要完成了这样几件事：

设置gcc的编译和连接选项，并将include/目录下的头文件添加到选项中。

将目录下的所有.c文件使用gcc编译为.o文件。

当使用make run指令时，Makefile文件还会使用gcc连接.o文件和动态库，生成可执行文件。可执行文件位于build/文件夹中，名称为nemu。

## 主要故障与调试

a.旧的不规范的代码导致的错误

在第一次实现push指令时，遇到的指令为push %ebp。由于寄存器压栈指令的机器码依此相邻，因而编写的原执行函数如下：

|  |
| --- |
| make\_EHelper(push) {  uint32\_t cc = decoding.opcode & 0x7;  id\_dest->val = id\_dest->width == 2 ? cpu.gpr[cc].\_16 : cpu.gpr[cc].\_32;  char reg\_name[10] = "%";  strcat(reg\_name, id\_dest->width == 2 ? regsw[cc] : regsl[cc]);  strcpy(id\_dest->str, reg\_name);  print\_asm\_template1(push);  } |

该函数根据机器码的操作码确定将哪一个寄存器压栈。但在后来，遇到将立即数压栈的指令时，因为没有意识到push需要修改，而发生了错误，调试了一段时间后才发现问题。

如果将原来的实行函数改写，对机器码的范围限定，超出限定的范围时终止程序并提示：

|  |
| --- |
| make\_EHelper(push) {  if(decoding.opcode >= 0x50 && decoding.opcode <= 0x57){  uint32\_t cc = decoding.opcode & 0x7;  id\_dest->val = id\_dest->width == 2 ? cpu.gpr[cc].\_16 : cpu.gpr[cc].\_32;  char reg\_name[10] = "%";  strcat(reg\_name, id\_dest->width == 2 ? regsw[cc] : regsl[cc]);  strcpy(id\_dest->str, reg\_name);  }  else TODO();  print\_asm\_template1(push);  } |

则不会出现长时间的调试过程。

b.CWTL指令实现错误

在80386的手册中，说明了CWTL指令的功能，将AX或AL中的值进行扩展，并传入EAX中。但因为实现错误，将结果写入了AX寄存器中。结果测试时发生故障，程序跳转到位置点而无法停止。

# **异常控制流**

## 功能实现的要点

(1) 实现系统调用

系统调用时通过自陷指令来实现的。在GNU/Linux中，用户程序通过自陷指令int $0x80指令触发系统调用，Nanos-lite也沿用这个约定。我们在x86-nemu的CTE中通过int $0x81实现\_yield()，虽然它们在CTE中引发了不同的事件，但从上下文保存到事件分发，它们的过程都是非常相似的。既然我们通过自陷指令来触发系统调用，那么对用户程序来说，用来向操作系统描述需求的最方便手段就是使用通用寄存器了，因为执行自陷指令之后，执行流就会马上切换到操作系统事先设置好的入口，通用寄存器也会作为上下文的一部分被保存起来。系统调用处理函数只需要从上下文中获取必要的信息，就能知道用户程序发出的服务请求是什么了。

Navy-apps已经为用户程序准备好了系统调用的接口了。navy-apps/libs/libos/src/nanos.c中定义的\_syscall\_()函数已经蕴含着上述过程：

intptr\_t \_syscall\_(int type，intptr\_t a0，intptr\_t a1，intptr\_t a2) {

int ret：

asm volatile("int $0x80"： "=a"(ret)： "a"(type)，"b"(a0)，"c"(a1)，"d"(a2))：

return ret：

}

上述内联汇编会先把系统调用的参数依次放入%eax，%ebx，%ecx，%edx四个寄存器中，然后执行自陷指令int $0x80。x86-nemu的CTE会将这个自陷操作打包成一个系统调用事件\_EVENT\_SYSCALL，并交由Nanos-lite继续处理。

Nanos-lite收到系统调用事件之后，就会调出系统调用处理函数do\_syscall()进行处理。do\_syscall()首先通过宏GPR1从上下文c中获取用户进程之前设置好的系统调用参数，通过第一个参数 - 系统调用号 - 进行分发。但目前Nanos-lite没有实现任何系统调用，因此触发了panic。

添加一个系统调用比你想象中要简单，所有信息都已经准备好了。我们只需要在分发的过程中添加相应的系统调用号，并编写相应的系统调用处理函数sys\_xxx()，然后调用它即可。回过头来看dummy程序，它触发了一个SYS\_yield系统调用。我们约定，这个系统调用直接调用CTE的\_yield()即可，然后返回0.

处理系统调用的最后一件事就是设置系统调用的返回值。我们约定系统调用的返回值存放在系统调用号所在的寄存器中，所以我们只需要通过GPRx来进行设置就可以了。

经过CTE，执行流会从do\_syscall()一路返回到用户程序的上述内联汇编中。内联汇编最后从%eax寄存器中取出系统调用的返回值，并返回给\_syscall()的调用者，告知其系统调用执行的情况(如是否成功等)。

(2) 堆区管理

malloc()/free()库函数，它们的作用是在用户程序的堆区中申请/释放一块内存区域。堆区的使用情况是由libc来进行管理的，但堆区的大小却需要通过系统调用向操作系统提出更改。这是因为，堆区的本质是一片内存区域，当需要调整堆区大小的时候，实际上是在调整用户程序可用的内存区域。事实上，一个用户程序可用的内存区域要经过操作系统的分配和管理的。

调整堆区大小是通过sbrk()库函数来实现的，它的原型是

void\* sbrk(intptr\_t increment);

用于将用户程序的program break增长increment字节，其中increment可为负数。所谓program break，就是用户程序的数据段(data segment)结束的位置。我们知道可执行文件里面有代码段和数据段，链接的时候ld会默认添加一个名为\_end的符号，来指示程序的数据段结束的位置。用户程序开始运行的时候，program break会位于\_end所指示的位置，意味着此时堆区的大小为0。malloc()被第一次调用的时候，会通过sbrk(0)来查询用户程序当前program break的位置，之后就可以通过后续的sbrk()调用来动态调整用户程序program break的位置了。当前program break和和其初始值之间的区间就可以作为用户程序的堆区，由malloc()/free()进行管理。注意用户程序不应该直接使用sbrk()，否则将会扰乱malloc()/free()对堆区的管理记录。

在Navy-apps的Newlib中，sbrk()最终会调用\_sbrk()，它在navy-apps/libs/libos/src/nanos.c中定义。框架代码让\_sbrk()总是返回-1，表示堆区调整失败，事实上，用户程序在第一次调用printf()的时候会尝试通过malloc()申请一片缓冲区，来存放格式化的内容。若申请失败，就会逐个字符进行输出。如果你在Nanos-lite中的sys\_write()中通过Log()观察其调用情况，你会发现用户程序通过printf()输出的时候，确实是逐个字符地调用write()来输出的。

但如果堆区总是不可用，Newlib中很多库函数的功能将无法使用，因此现在你需要实现\_sbrk()了。为了实现\_sbrk()的功能，我们还需要提供一个用于设置堆区大小的系统调用。在GNU/Linux中，这个系统调用是SYS\_brk，它接收一个参数addr，用于指示新的program break的位置。\_sbrk()通过记录的方式来对用户程序的program break位置进行管理，其工作方式如下：

a.program break一开始的位置位于\_end

b.被调用时，根据记录的program break位置和参数increment，计算出新program break

c.通过SYS\_brk系统调用来让操作系统设置新program break

d.若SYS\_brk系统调用成功，该系统调用会返回0，此时更新之前记录的program break的位置，并将旧program break的位置作为\_sbrk()的返回值返回

f.若该系统调用失败，\_sbrk()会返回-1

上述代码是在用户层的库函数中实现的，我们还需要在Nanos-lite中实现SYS\_brk的功能。由于目前Nanos-lite还是一个单任务操作系统，空闲的内存都可以让用户程序自由使用，因此我们只需要让SYS\_brk系统调用总是返回0即可，表示堆区大小的调整总是成功。

(3) 虚拟文件系统

为了实现一切皆文件的思想，我们需要扩展之前实现的文件操作：我们不仅需要对普通文件进行读写，还需要支持各种"特殊文件"的操作。至于扩展的方式，你是再熟悉不过的了，那就是抽象!

我们对之前实现的文件操作API的语义进行扩展，让它们可以支持任意文件(包括"特殊文件")的操作:

int fs\_open(const char \*pathname，int flags，int mode);

ssize\_t fs\_read(int fd，void \*buf，size\_t len);

ssize\_t fs\_write(int fd，const void \*buf，size\_t len);

off\_t fs\_lseek(int fd，off\_t offset，int whence);

int fs\_close(int fd);

这组扩展语义之后的API有一个酷炫的名字，叫[VFS(虚拟文件系统)](https://en.wikipedia.org/wiki/Virtual_file_system)。既然有虚拟文件系统，那相应地也应该有"真实文件系统"，这里所谓的真实文件系统，其实是指具体如何操作某一类文件。比如在Nanos-lite上，普通文件通过ramdisk的API进行操作; 在真实的操作系统上，真实文件系统的种类更是数不胜数：比如熟悉Windows的你应该知道管理普通文件的NTFS，目前在GNU/Linux上比较流行的则是EXT4; 至于特殊文件的种类就更多了，于是相应地有procfs，tmpfs，devfs，sysfs，initramfs..。这些不同的真实文件系统，它们都分别实现了这些文件的具体操作方式。

所以，VFS其实是对不同种类的真实文件系统的抽象，它用一组API来描述了这些真实文件系统的抽象行为，屏蔽了真实文件系统之间的差异，上层模块(比如系统调用处理函数)不必关心当前操作的文件具体是什么类型，只要调用这一组API即可完成相应的文件操作。有了VFS的概念，要添加一个真实文件系统就非常容易了：只要把真实文件系统的访问方式包装成VFS的API，上层模块无需修改任何代码，就能支持一个新的真实文件系统了。

在Nanos-lite中，实现VFS的关键就是Finfo结构体中的两个读写函数指针:

typedef struct {

char \*name; // 文件名

size\_t size; // 文件大小

off\_t disk\_offset; // 文件在ramdisk中的偏移

off\_t open\_offset; // 文件被打开之后的读写指针

ReadFn read; // 读函数指针

WriteFn write; // 写函数指针

} Finfo;

其中ReadFn和WriteFn分别是两种函数指针，它们用于指向真正进行读写的函数，并返回成功读写的字节数。有了这两个函数指针，我们只需要在文件记录表中对不同的文件设置不同的读写函数，就可以通过f->read()和f->write()的方式来调用具体的读写函数了。

不过在Nanos-lite中，由于特殊文件的数量很少，我们约定，当上述的函数指针为NULL时，表示相应文件是一个普通文件，通过ramdisk的API来进行文件的读写，这样我们就不需要为大多数的普通文件显式指定ramdisk的读写函数了。

## 必答题

a.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。

答：读取游戏存档：程序通过fread调用了read系统调用，操作系统（Nanos-lite）触发读取文件操作。因为该文件为普通文件（放置于硬盘），因而使用ramdisk\_read()函数完成fread操作。

由于该硬盘为虚拟硬盘，实际上装载在了AM的相关内存位置，因而ramdisk\_read函数调用了AM的memcpy函数，将需求信息在内存中进行交换。而memcpy函数运行在NEMU的x86虚拟机上，通过执行mov的汇编指令完成。

而屏幕的更新，有NDL\_DrawRect函数开始，通过写/dev/fb文件，调用了libos中的\_write函数，而该函数调用系统调用SYS\_write，读写虚拟文件/dev/fb。因为/dev/fb不是普通文件，因而使用的写函数fb\_write是对设备的操作。该操作调用AM设备的写函数。在AM中，我们使用内存映射I/O完成显示器的写入，因而完成相关内存映射后，可以直接在NEMU中用数据转移的汇编指令完成更新屏幕的操作。

## 主要故障与调试

a.在完成系统调用的过程中，很重要的一点，保证iret和int指令的栈操作顺序正确。因为该栈操作涉及程序运行的上下文。

当执行int指令时，主要需要执行下面操作：

依次将EFLAGS, CS(代码段寄存器), EIP寄存器的值压栈；从IDTR中读出IDT的首地址；根据异常号在IDT中进行索引, 找到一个门描述符；将门描述符中的offset域组合成目标地址；跳转到目标地址。

在int指令后，在中断的运行程序中，我们一般保存错误码和异常号#irq，把当前的通用寄存器保存到栈上，并通过一条pushl $0指令在栈上占位，它是为PA4准备的。些内容连同之前保存的错误码, #irq，以及硬件保存的EFLAGS, CS, EIP，形成了完整的上下文。

为了维护正确的上下文，我们将压栈的内容组织为一个\_Context的结构体。在刚开始，我的结构体组织形况为（该结构体为压栈顺序的逆序）：

|  |
| --- |
| struct \_Context {  uintptr\_t edi, esi, ebp, esp, ebx, edx, ecx, eax;  int irq;  uintptr\_t err, eip, cs, eflags;  struct \_Protect \*prot;  }; |

结果发现产生偏差。后来，发现是因为push指令存在问题。Push指令先将esp寄存器减4后压栈，但在NEMU的实现中，如果压栈esp寄存器，则会将减法前的寄存器压入栈中。

后来对\_Context进行修改，改为：

|  |
| --- |
| struct \_Context {  uintptr\_t not\_care;  uintptr\_t edi, esi, ebp, esp, ebx, edx, ecx, eax;  int irq;  uintptr\_t err, eip, cs, eflags;  struct \_Protect \*prot;  }; |

程序运行正常。

# **异常控制流**

## 课设总结

## 课设心得