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：	U201514631
姓　　名：	王晶
电　　话：	17786139970
邮　　件：	792732547@qq.com
完成日期：	2017-10-31 周二上午

计算机科学与技术学院

目 录

•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: _____

1　课程设计概述

1.1 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。

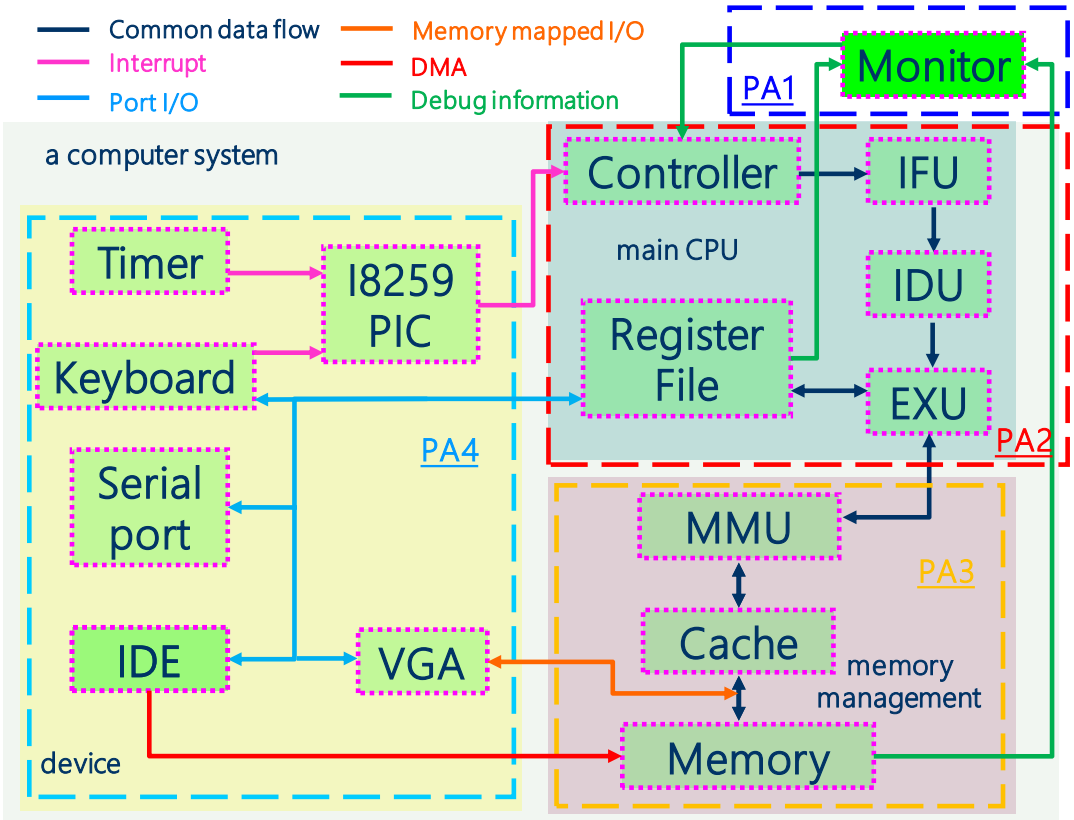

图 1.1 NEMU 模拟器计算机框架示意图

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。

1.2 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。

	持续时间/周	预计耗时/小时	代码量/行
PA0 – 开发环境配置	0.2	10	无
PA1 – 简易调试器	0.8	30	400
PA2 – 指令系统	1	60	800
PA3 – 存储管理	1	50	500
PA4 – 中断与I/O	1	30	300
总计	4	180	2000

图 1.2 设计任务

2 PA1 简易调试器

2.1 功能实现的要点

2.1.1 阶段 1

该阶段包括实现正确的寄存器结构、单步执行、打印寄存器及扫描内存的功能。

在 CPU_state 结构体中，目前包含 8 个 32 位的通用寄存器与 1 个指令指针寄存器 eip。为了能按读取的寄存器的不同字，并能根据寄存器名访问相应寄存器，可以采用匿名 union 结合匿名 struct 方式实现。

单步执行、打印寄存器及扫描内存的功能只需调用 reg.h/c 和 memory.h/c 相应接口完成。

2.1.2 阶段 2——表达式求值

采用简易编译器的思想，对表达式进行分析并求值。首先通过 make_tokens 函数对表达式进行词法分析。词法分析正确后，通过 check_parentheses 函数检查括号是否匹配及正确，再通过 eval 函数对表达式进行递归求值。eval 函数采用了分治的思想，每次先找出当前子表达式中的主运算符，对运算符两侧的表达式递归求值。

对负号以及取地址符的识别，可在词法分析结束后，扫描一遍 tokens，若某个‘-’或‘*’前不是一个操作数，则说明该标记是负号或者取地址符。

对于表达式计算的检测，采取了借用编译器帮助计算的巧妙思想。将表达式写入一个程序源代码，编译运行程序自然能得到该表达式的结果。

2.1.3 阶段 3——监视点

监视点采用链表实现。维护一个使用的监视点链表以及一个空闲监视点链表。在 cpu_exec 函数中，每次执行完一条指令，便检查监视点链表，若其中有值发生改变，则中断程序的执行。

2.2 必答题

2.2.1 理解基础设施

调试上花费的时间：

500 × 90% × 20 × 30s = 270000 seconds = 4500 minutes = 75 hours

2.2.2 查阅 i386 手册

1. EFLAGS 寄存器中的 CF 位是什么意思？

（2.3.4 节 Flags Register）

由 EFLAGS Register 图中可知，CF 表示 CARRY FLAG，即进位标志。附录 C 中叙述了 CF 位的详细内容：

Carry Flag -- Set on high-order bit carry or borrow; cleared otherwise.

2. ModR/M 字节是什么？

（17.2.1 节 ModR/M and SIB Bytes）

ModR/M 是指令格式中跟在 OPCODE 后的字节。它包含 3 个域的信息：

（1） mod field：占据 2 位，与 r/m 域结合组成 32 种可能的值：8 个寄存器和 24 个索引模式。

（2） reg field：占据 3 位，要么指示一个寄存器号，要么表示 opcode 更多信息。

（3） r/m field：占据 3 位，指示操作数地址的寄存器，或和 mod field 组合描述寻址方式。

3. mov 指令的具体格式是怎么样的？

（17.2.2.11 节 Instruction Set Detail 中的 MOV 部分）

MOV 指令格式如图 2.1 所示。

Opcode	Instruction	Clocks	Description
88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte
89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword
8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
8B /r	MOV r32,r/m32	2/4	Move r/m dword to dword register
8C /r	MOV r/m16,Sreg	2/2	Move segment register to r/m word
8D /r	MOV Sreg,r/m16	2/5,pm=18/19	Move r/m word to segment register
A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	2	Move AL to (seg:offset)
A3	MOV moffs16,AX	2	Move AX to (seg:offset)
A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)
B0 + rb	MOV reg8,imm8	2	Move immediate byte to register
B8 + rw	MOV reg16,imm16	2	Move immediate word to register
B8 + rd	MOV reg32,imm32	2	Move immediate dword to register
Ciiiiii	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
C7	MOV r/m16,imm16	2/2	Move immediate word to r/m word
C7	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

图 2.1 MOV 指令格式

2.2.3 shell 命令

1. 统计 nemu/目录下的所有.c 和.h 文件总共有多少行代码

使用命令：find -name '*.[ch]' | xargs wc -l

得到结果 4264 行。

2. 在 PA1 中编写了多少行代码？

使用 git checkout pa0 回到 pa1 工作之前，使用上述命令得到结果 3825 行，故我在 PA1 中编写了 439 行代码。

3. 除去空行统计代码行数

使用命令：find -name '*.[ch]' | xargs cat | grep -v ^$ | wc -l

得到结果 3526 行。

2.2.4 使用 man

1. 请解释 gcc 中的-Wall 和-Werror 有什么作用？

使用 man gcc 命令查看关于 gcc 的手册。-Wall 启用所有警告，-Werror 使所有警告变成错误。

2. 为什么要使用-Wall 和-Werror？

-Wall 和-Werror 同时开启可将所有可能的警告变成错误而使得编译失败，这样做的好处是可以通过编译器来帮助排除所有可能导致错误的地方。

2.3 主要故障与调试

故障 1：多次计算表达式结果会出错。如先计算“12＋3”得到 15，再计算“1＋3”得到 15。

故障原因：保存 tokens 的数组重新使用时没有清空，还保存了上一次的数据，故第二个表达式第一个操作数虽然是“1”，但由于 token 中第二个字节还残留了‘2’。

解决方法：每次填写一个 token 向其后填写一个’\0’。同时，需要检测每个 token 的长度不能超过保存其的字符数组的大小。

3 PA2 冯诺依曼计算机系统

3.1 功能实现的要点

3.1.1 阶段 1——运行第一个客户程序

任务：需要实现 call、push、pop、sub、xor、ret 指令，以及在寄存器结构体中添加 EFLAGS 寄存器。能够正确运行“dummy”测试程序。

EFLAGS 寄存器，由于每个标志位只有 1 位，故使用位域实现。

实现指令前，需要先仔细阅读代码，理清 nemu 实现对指令的取指、译码以及实现的流程，以及代码框架中提供的一些函数的作用。还需要掌握查阅 i386 手册的方法。

call 指令调用跳转译码函数 decode_J，执行函数中，需要先将当前 eip 入栈再跳转。

ret 指令则先将 eip 出栈，再跳转回该地址。

sub、xor 指令需要相应设置标志位。

3.1.2 阶段 2——程序，运行时环境与 AM

任务：继续实现更多指令，通过一系列测试。实现常用的库函数，包括字符串处理函数、sprintf 等。实现 Differential Testing。

实现更多指令的要点即依据 i386 手册进行实现。可以采用 KISS 原则，依次运行测试程序，根据需要来补充指令。

根据反汇编代码得到的机器指令，可以快速得知需要补充的指令。调试时可以将“nexus-am/am/arch/x86-nemu/img”路径下的 run 脚本中的“-b”参数删去，即可使用 pa1 实现的简易调试器进行调试，结合反汇编代码，能够较快找出错误。

实现 sprintf 函数，需要结合可变参数。本质上是解析字符串。目前仅需实现“%s”和“%d”的功能。

Differential Testing 调用 qemu 运行 x86-nemu 的 AM 程序，并与 nemu 进行比对。每执行一条指令，便对比 8 组通用寄存器和 eip 寄存器的内容。为此，需要保证 nemu 中的寄存器顺序与 qemu 参照中的一致。阅读源代码，找到 qemu-diff 中的寄存器顺序，其位置在“nemu/tools/qemu-diff/include/protocol.h”中，如图 3.1 所示。

```
union gdb_regs {
  struct {
    uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    uint32_t eip, eflags;
    uint32_t cs, ss, ds, es, fs, gs;
  };
  struct {
    uint32_t array[77];
  };
};
```

图 3.1 qemu-diff 的寄存器顺序

3.1.3 阶段 3——输入输出

任务：实现 IO 功能。实现 in、out 指令，实现 printf 函数，实现时钟、键盘、VGA 的设备 IO，通过跑分测试。

printf 与 sprintf 主体功能大致相同，故可将它们公共的部分提取出来，避免代码重复。

实现 IO 需要阅读代码，理解 nemu IO 的原理。可以对比 native 中的相应的 IO 部分，仿照实现，完成相同的功能。

VGA 的设备 IO 为内存映射，为提高效率，应使用 memcpy 函数进行内存拷贝，且 memcpy 函数的实现应优先以 4 字节（或 8 字节）单位进行拷贝。

x86-nemu 运行 MicroBench 跑分结果如图 3.2 所示。

```
[dinic] Dinic's maxflow algorithm: * Passed.
  min time: 1862 ms [726]
[lzip] Lzip compression: * Passed.
  min time: 5143 ms [514]
[ssort] Suffix sort: * Passed.
  min time: 1031 ms [573]
[md5] MD5 digest: * Passed.
  min time: 10430 ms [187]
===============================================
MicroBench PASS        453 Marks
                  vs. 100000 Marks (i7-6700 @ 3.40GHz)
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 3.2 x86-nemu MicroBench 跑分结果

native 运行 MicroBench 跑分结果如图 3.3 所示。

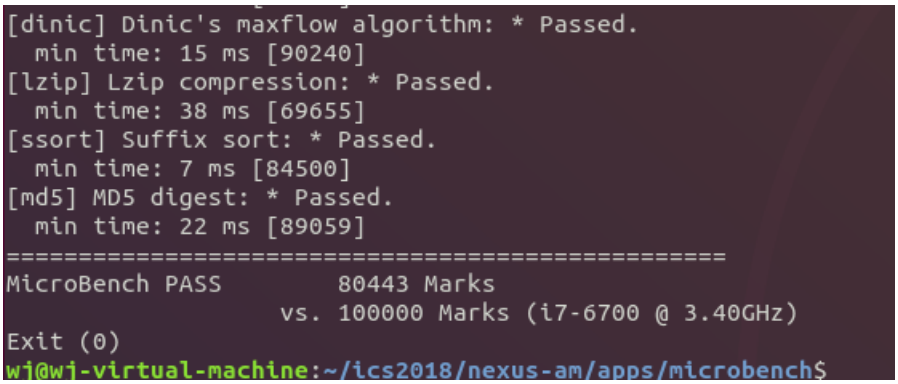

图 3.3 native MicroBench 跑分结果

3.2 必答题

3.2.1 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：1. 去掉 interpret_rtl_j 的 static，编译正确，未发生错误。

2. 去掉 inline 关键字，编译结果如图 3.4，发生错误，提示函数 interpret_rtl_j 定义了但未使用。根据错误提示信息，在 arith.c 文件中，包含了 rtl.h，即定义了函数，但未使用，故发生错误。

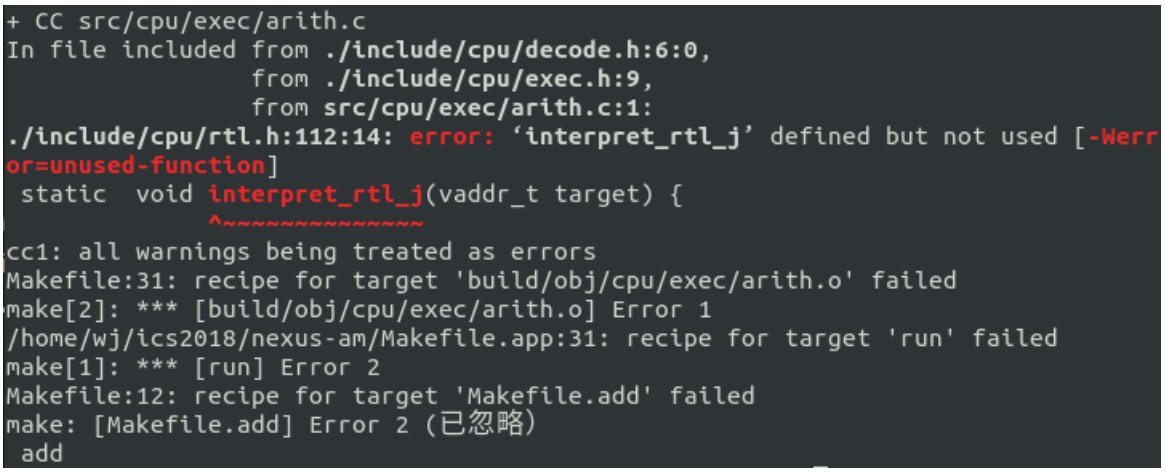

图 3.4 去掉 static 编译结果

3. 去掉 static 和 inline，编译结果如图 3.5 所示，发生错误，函数 interpret_rtl_j 被多次定义。因为在 rtl.h 中定义了该函数，而 rtl.h 被多个文件 include，故编译时发生多次定义错误。

```
build/obj/cpu/exec/system.o：在函数‘interpret_rtl_j’中：
/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：`interpret_rtl_j'被多次定义
build/obj/cpu/exec/arith.o:/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：第一次
在此定义
build/obj/cpu/exec/control.o：在函数‘interpret_rtl_j’中：
/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：`interpret_rtl_j'被多次定义
build/obj/cpu/exec/arith.o:/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：第一次
在此定义
build/obj/cpu/exec/special.o：在函数‘interpret_rtl_j’中：
/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：`interpret_rtl_j'被多次定义
build/obj/cpu/exec/arith.o:/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：第一次
在此定义
build/obj/cpu/exec/exec.o：在函数‘interpret_rtl_j’中：
/home/wj/ics2018/nemu/./include/cpu/rtl.h:113：`interpret_rtl_j'被多次定义
```

图 3.5 去掉 static 和 inline 编译结果

3.2.2 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？你是如何得到这个结果的?

答：31 个。使用命令“readelf -s build/nemu | grep dummy | wc -l”查看 nemu 符号表中 dummy 出现的次数。结果为 31。

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：使用命令“readelf -s build/nemu | grep dummy | wc -l”统计，结果为 31，与上题中相同。因为两次 dummy 均没有赋值，故其中一个变量 dummy 被当做声明，另一个被当做定义。

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：编译结果如图 3.6 所示，编译发生错误，dummy 重定义。因为之前 dummy 并

没有初始化赋值，属于弱定义。初始化后属于强定义，编译器不允许重定义。

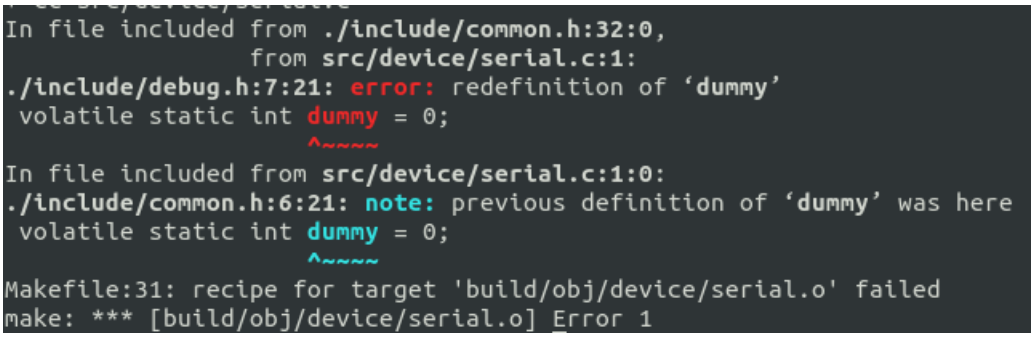

图 3.6 dummy 初始化后编译结果

3.2.3 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：工作方式：输入 make 命令后，make 会在当前目录找 Makefile 的文件。找到后，读入被 include 的其他文件，初始化文件中的变量，推导分析隐晦规则，为所有目标文件创建依赖关系链，根据依赖关系，决定哪些目标要重新生成，最后执行生成命令。

编译连接：make 根据最终目标文件依次递归查找需要的依赖文件，检测是否最新等。如当试图生成目标文件.o 时，会寻找依赖的.h 或.c 文件。

3.3 主要故障与调试

故障 1：在阶段 2 中，测试文件“goldbach”时，发生段错误。

调试发现问题：将文件“nemu-am/am/arch/x86-menu/img/run”中的“-b”参数去掉，可以使用 PA1 中实现的简易调试器进行调试。使用单步执行，发现程序在运行到“test”指令后发生段错误。对比对应的反汇编代码，发现自己程序中该指令缺少一个立即数操作数。核对指令与 i386 手册后，发现原因是该指令前缀为“f6 c1”，即 group3 中的 test，该 test 还需要读取一个立即数。

解决方法：在 group3 中的 test 再设置一个译码函数“decode_test_I”。

故障 2：在阶段 2 中，测试文件“recursion”发生错误（程序陷入死循环）。

调试发现问题：进行单步调试，与相应的反汇编代码进行对比。发现程序 call 指令“ff 15 e4 01 10 00”指令读取正确，跳转地址不同。查阅 i386 手册，发现该指令是跳转到立即数地址指向的内容，与我填写的执行函数“call”不符。

解决方法：将执行函数改为“call_rm”，即能使程序正确跳入立即数所指向的地址。

4　PA3 批处理系统

4.1 功能实现的要点

4.1.1 阶段 1——上下文切换

任务：实现 i386 中断机制；重新组织_Context 结构体；实现正确的事件分发；恢复上下文。

int 指令通过调用 raise_intr 函数完成，首先需要依次入栈 eflags、cs、eip 寄存器，然后跳转到中断号相应的执行函数地址。中断执行函数地址通过 IDTR 寄存器开始索引。

为了组织正确的_Context 结构体，需要研读代码，理清计算机是如何实现中断机制的。首先程序触发中断时，操作系统层会通过_yield 函数（一行汇编代码“int $0x81”）执行 int 指令，通过 raise_intr 函数先入栈了 eflags、cs 和 eip 寄存器；随后跳入了 81 号中断执行函数“vectrap”，该函数中入栈了 0 和 0x81，即错误代码和中断号；之后跳入了“asm_trap”中，执行了“pusha”入栈了 8 个通用寄存器、0（保护地址空间指针）和 esp，最后调用 irq_handle 进行中断处理。用流程图表示如图 4.1。

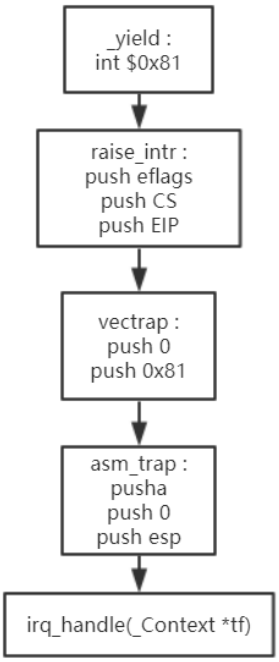

图 4.1 中断处理流程

最后入栈的 esp 为指示 irq_handle 的上下文参数所用，故其之上入栈的内容为上下文_Context 结构体的内容。综上，上下文结构体的内容依次为保护地址空间指针、8 个通用寄存器（逆序）、中断号、错误代码、eip、cs、eflags。

4.1.2 阶段 2——用户程序与系统调用

任务：实现 loader；识别系统调用；实现 SYS_yield 系统调用；实现 SYS_exit 系统调用；在 Nanos-lite 上运行 Hello world；实现堆区管理；

识别与实现系统调用，实现正确的系统调用分发，并实现相应的系统调用函数 sys_xxx 即可。

实现堆区管理，目前只需要在用户层通过记录管理 program_break 即可，操作系统层直接返回 0 表示堆区大小调整成功。

4.1.3 阶段 3——文件系统与批处理系统

任务：让 loader 使用文件；实现完整的文件系统；把串口抽象成文件；把 VGA 显存抽象成文件；把设备输入抽象成文件；在 NEMU 中运行仙剑奇侠传；自由开关 DiffTest 模式；快照；添加开机菜单与批处理系统。

首先实现文件的基本操作函数 open、read、write、close、lseek。需要注意的点是维护文件打开后的读写指针 open_offset，并且注意文件读写不能超过边界。

将设备虚拟成文件，只需要实现针对设备的读写函数，并在文件列表中设置相应读写函数。

运行仙剑奇侠传时，我的程序报错遇到未实现的指令“stos”。通过查看 x86-nemu 与 native 的反汇编代码，发现该 stos 指令出现在仙剑奇侠传的 PAL_InitInput 函数中，原本是调用 memset 函数进行初始化，然而反汇编代码没有调用而是直接通过指令进行，故此为编译器优化的结果，实现 stos 后仙剑奇侠传正确运行。

自由开关 DiffTest，需要实现 detach 与 attach 功能。attach 时需要将 nemu 当前的寄存器内容与内存中的内容拷贝给 qemu。其中值得注意的点是，DiffTest 中用到的一些函数均为 static 函数，不能在其他文件中使用，解决方法是 DiffTest 提供一个接口给

ui.c，让 ui.c 调用该接口，在 DiffTest 文件中实现上述功能。快照功能只需将当前寄存器与内存中的内容写入文件，加载快照时从文件读取恢复寄存器与内存。

实现批处理系统，需要实现 SYS_execve 系统调用。

4.2 必答题

4.2.1 文件读写的具体过程

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.

PAL_LoadGame()中，通过 fread()读取游戏存档文件。首先通过 fopen()函数打开，fopen 函数调用_fopen_r，_fopen_r 调用_open_r。libc 中的_open_r 调用 libos 中的_open 函数，其位于 navy-apps/libs/libos/src/nanos 中。_open 通过_syscall_（“int $0x80”）进行系统调用，进入 Nanos-lite 中，通过 fs_open 打开文件。之后通过运行时环境 AM 提供的接口完成指定功能，而运行时环境依赖于 NEMU 提供的底层硬件模拟实现。

fread 库函数定义在 navy-apps/libs/libc/src/stdio/fread.c 中，最终通过定义在 navy-apps/libs/libc/src/string/memcpy.c 中的 memcpy 函数完成。

redraw()函数调用 NDL_DrawRect 函数，NDL_DrawRect 函数调用 fwrite 函数进行写，而文件为 stdout 即标准输出。fwrite 调用_fwrite_r，_fwrite_r 调用__sputc_r，__sputc_r 调用__swbuf_r，__swbuf_r 调用_fflush_r，_fflush_r 调用__sflush_r，__sflush_r 调用了_write 系统调用。通过系统调用进入 Nanos-lite，通过 fs_write 完成写操作。该过程与 fopen 类似，只是调用关系更复杂一些。

综上，用户程序（仙剑奇侠传）、库函数（libc）、libos、操作系统（Nanos-lite）、运行时环境（AM）、底层硬件（Nemu）相互协助，完成了用户程序需求的功能。

4.3 主要故障与调试

故障 1：在完成实现堆区管理阶段，hello 程序只能输出 9 次 helloworld，程序便出现段错误崩溃。为此，在_sbrk 函数中，借助 sprintf 调试，sprintf 使用“%d”参数也会导致段错误。

调试发现问题：启用简易调试器进行调试，同时开启 DiffTest，并同时对比 hello 程序的反汇编代码。DiffTest 显示在一条 jne 指令，nemu 与 qemu 发生了区别。而该跳转指令的前一条为 shr 指令。即 shr 指令执行后，寄存器内结果相同，但 jne 跳转的结果却不同。由于 jne 是根据标志位进行跳转的，故检查标志位。发现在阶段 2 实现移位指令时，我没有设置标志位。

解决方法：修改移位指令，按 i386 手册上的说明相应设置标志位。

故障 2：运行仙剑奇侠传时，提示 stos 指令未实现。

发现问题：该指令出现在 PAL_InitInput 函数中，调用 memset 函数的功能被编译成若干条指令实现。通过询问其他同学以及查看其他同学的反汇编代码，发现有同学的反汇编代码并没有优化而是直接 call memset。故考虑此为不同编译器版本处理的结果。

解决方法：实现 stos 指令。