

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： 2018-12-29 周六上午



计算机科学与技术学院

目 录

1	课程设计概述.....	1
1.1	课设背景	1
1.2	设计任务	2
2	PA1 简易调试器.....	3
2.1	功能实现的要点.....	3
2.2	必答题	5
2.3	主要故障与调试.....	9
3	PA2 冯诺依曼计算机系统.....	10
3.1	功能实现的要点.....	10
3.2	必答题	13
3.3	主要故障与调试.....	16
4	PA3 批处理系统.....	17
4.1	功能实现的要点.....	17
4.2	必答题	19
4.3	主要故障与调试.....	20
5	PA4 分时多任务.....	22
5.1	功能实现的要点.....	22
5.2	必答题	23
5.3	主要故障与调试.....	24
6	总结与体会.....	26

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。

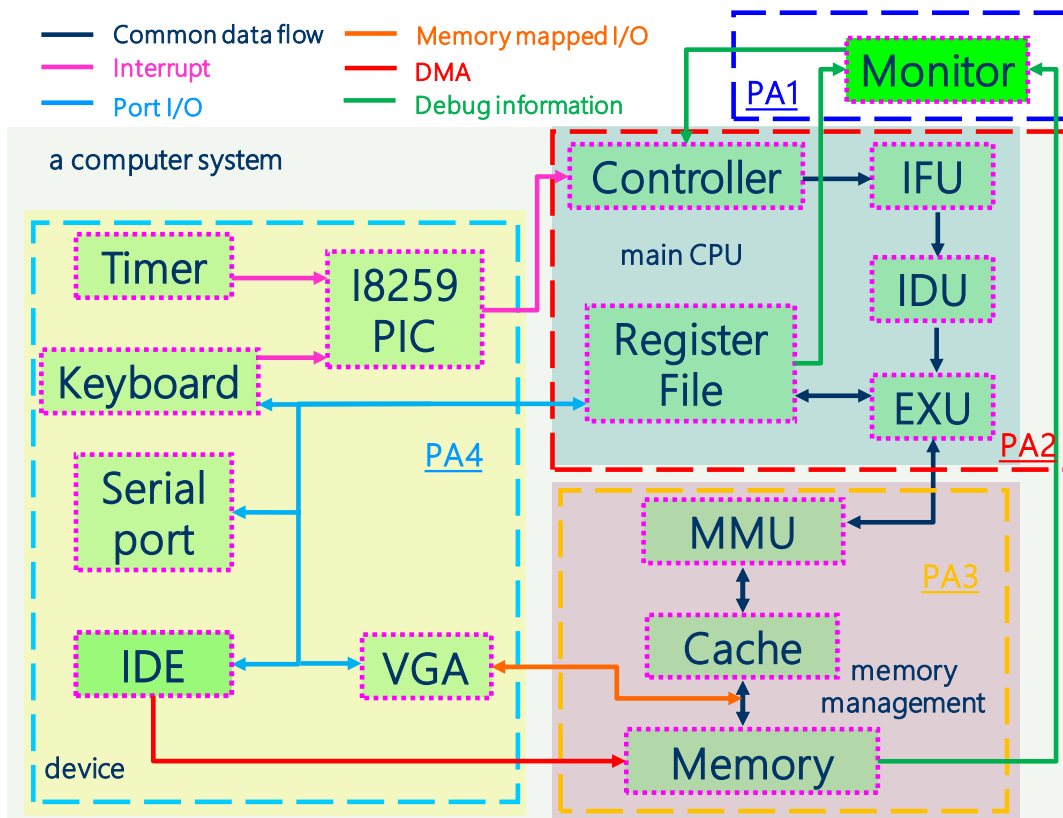


图 1.1 NEMU 模拟器计算机框架示意图

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，如图 2.1。

采用匿名 `union`，每个 `eax` 占据 32 位的空间，`_32`、`_16`、`_8` 共享 32 位地址空间，实现了按不同字节读取寄存器的功能。

```
union {
    union {
        uint32_t _32;
        uint16_t _16;
        uint8_t _8[2];
    } gpr[8];

    /* Do NOT change the order of the GPRs' definitions. */

    /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
     * in PA2 able to directly access these registers.
     */
    struct {
        rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
};
```

图 2.1 通用寄存器

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

该阶段任务：实现算术表达式的词法分析；实现算术表达式的递归求值；实现表达式生成器。

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，为此需要为算术表达式的各种 `token` 类型添加规则。成功识别出 `token` 后，将 `token` 的信息依次记录到 `tokens` 数组中。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。

分析主运算符时，可从右至左结合优先级扫描运算符。

表达式生成器，采取了借用编译器计算表达式的巧妙思想，来帮助检查表达式求值的结果。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

该阶段任务：扩展表达式求值的功能；实现监视点池的管理；实现监视点。

扩展表达式求值的功能，添加逻辑运算符 “==”、“!=” 和 “&&” 等，仅需在 `eval` 中添加相应的 `case` 并注意与其他运算符的优先级关系。

实现监视点池的管理，维护一个使用的监视点链表以及一个空闲监视点链表，一个监视点结构中需要保存该监视点的表达式、监视点编号以及上一次运算的结果。

实现监视点，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

- 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?

调试上花费的时间:

$$500 \times 90\% \times 20 \times 30s = 270000 \text{ seconds} = 4500 \text{ minutes} = 75 \text{ hours}$$

采用简易调试器节省时间:

$$500 \times 90\% \times 20 \times (30 - 10) s = 180000 \text{ seconds} = 50 \text{ hours}$$

2.2.2 查阅 i386 手册

- 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?

1. EFLAGS 寄存器中的 CF 位是什么意思?

(2.3.4 节 Flags Register)

由 EFLAGS Register 图 2.2 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.

Figure 2-8. EFLAGS Register

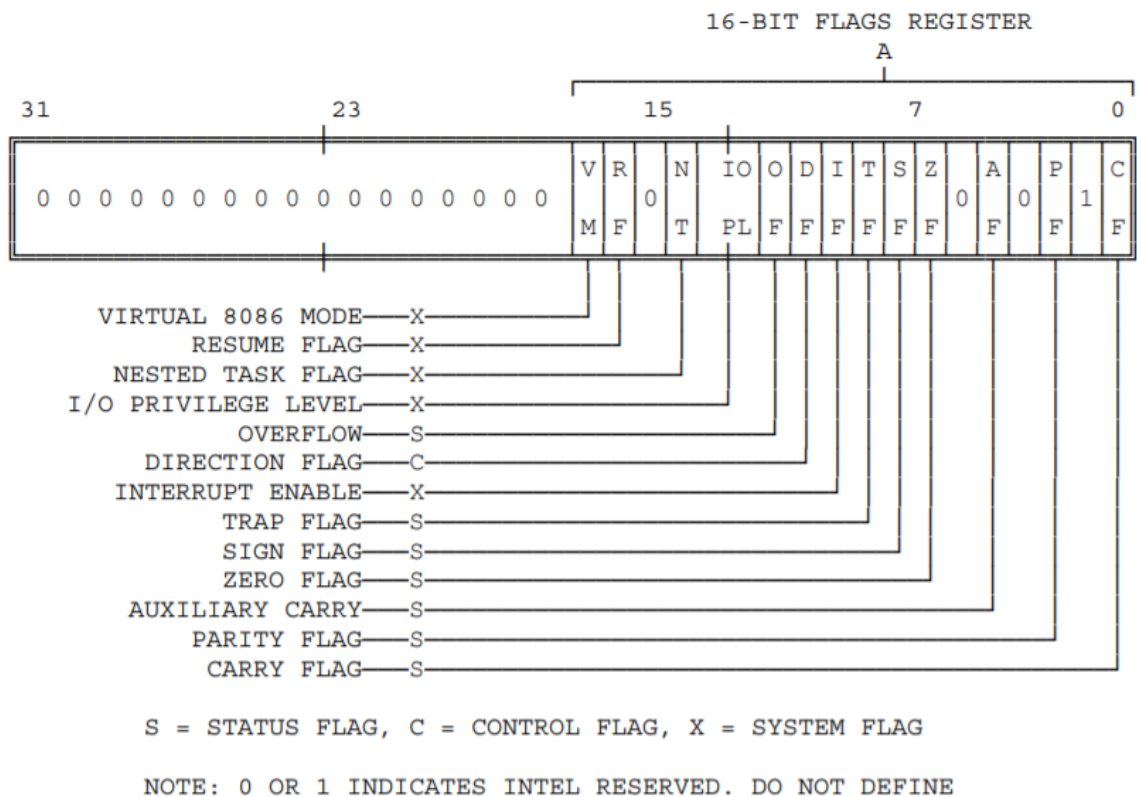


图 2.2 EFLAGS 寄存器

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 2.3 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 2.3 MOV 指令格式

2.2.3 shell 命令

- **shell命令** 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? 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“过去”?) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地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?

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。

在 `Makefile` 中添加 `count` 选项, 如图 2.4 所示, 如此可以敲入 “`make count`” 就可以自动运行统计代码行数的命令。

```
count:
    find -name '*.ch]' | xargs wc -l
```

69,34-41 底端

图 2.4 make count 命令

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

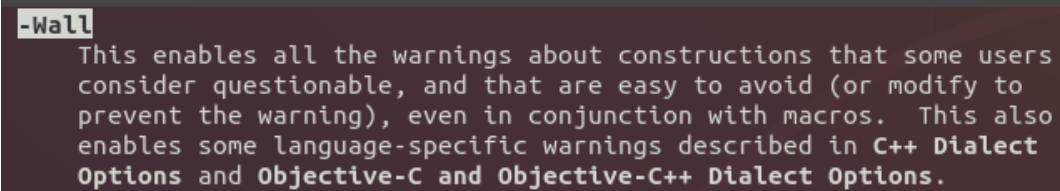
使用命令：`find -name '*.ch' | xargs cat | grep -v ^$ | wc -l`

得到结果 3526 行。

2.2.4 使用 man

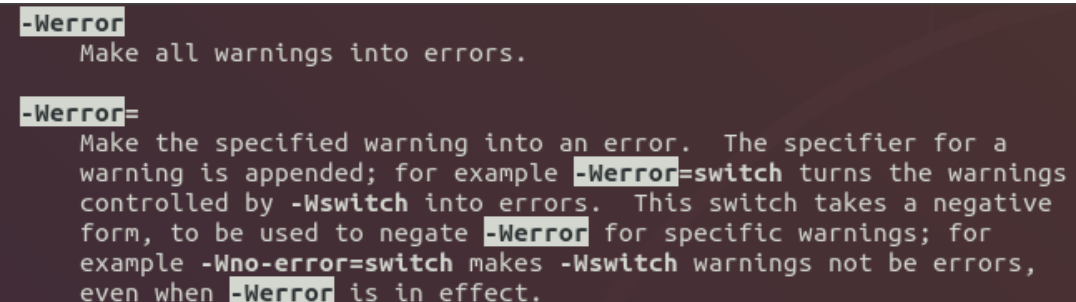
- 使用 `man` 打开工程目录下的 `Makefile` 文件，你会在 `CFLAGS` 变量中看到 `gcc` 的一些编译选项。请解释 `gcc` 中的 `-Wall` 和 `-Werror` 有什么作用？为什么要使用 `-Wall` 和 `-Werror` ？

1. 请解释 `gcc` 中的 `-Wall` 和 `-Werror` 有什么作用？



```
-Wall
This enables all the warnings about constructions that some users
consider questionable, and that are easy to avoid (or modify to
prevent the warning), even in conjunction with macros. This also
enables some language-specific warnings described in C++ Dialect
Options and Objective-C and Objective-C++ Dialect Options.
```

图 2.5 `gcc -Wall` 选项



```
-Werror
Make all warnings into errors.

-Werror=
Make the specified warning into an error. The specifier for a
warning is appended; for example -Werror=switch turns the warnings
controlled by -Wswitch into errors. This switch takes a negative
form, to be used to negate -Werror for specific warnings; for
example -Wno-error=switch makes -Wswitch warnings not be errors,
even when -Werror is in effect.
```

图 2.6 `gcc -Werror` 选项

使用 `man gcc` 命令查看关于 `gcc` 的手册。查询 `-Wall` 和 `-Werror` 选项，如图 2.5 和图 2.6 所示。`-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 (Keep it Simple and Stupid) 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 `pal` 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。由于 `printf` 与 `sprintf` 功能基本一致，只是将字符串输出的目标地址不同，故可将字符串处理的功能提取出来作为一个函数 `vsprintf`，`sprintf` 和 `printf` 调用 `vsprintf`，减少代码冗余。

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

对于捕捉死循环，可以检测各个跳转指令执行的次数。因为死循环必然有跳转指令，包括条件跳转与函数调用等，只需要在 nemu 中统计每条跳转指令执行的次数，当某条跳转指令执行次数大于一个较大的阈值（如 10e6）时，即可认为发生了死循环。

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。

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。

阅读代码可知，设备 IO 读写功能由 IO 封装后提供给 Nanos-lite。时钟、键盘以及 VGA 均抽象成 _Device 结构，并通过结构中的读写函数指针调用设备相应的 IO 函数。时钟的读取、键盘输入的读取以及 VGA 设备宽和高的读取都是端口 IO，VGA 绘制画面使用的是内存映射 IO。

以 VGA 设备读取为例，NEMU 在 vga.c 中，通过 init_vga 函数将屏幕大小暴露给 CPU，代码如图 3.2 所示。根据此，可以得知 AM 中应从 SCREEN_PORT 读取 4 字节数据，其中高 16 位为宽度，低 16 位为高度。

```
screensize_port_base = add_pio_map(SCREEN_PORT, 4, NULL);  
*screensize_port_base = ((SCREEN_W) << 16) | (SCREEN_H);
```

图 3.2 VGA 设备暴露屏幕大小给 CPU

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，一次对一行的内容进行拷贝，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 3.3 所示，得分为 453 分。

```
[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 3.3 x86-nemu MicroBench 跑分结果

native 运行 MicroBench 跑分结果如图 3.4 所示，得分为 80443 分。

```
[dinic] Dinic's maxflow algorithm: * Passed.  
min time: 15 ms [90240]  
[lzip] Lzip compression: * Passed.  
min time: 38 ms [69655]  
[ssort] Suffix sort: * Passed.  
min time: 7 ms [84500]  
[md5] MD5 digest: * Passed.  
min time: 22 ms [89059]  
=====
```

MicroBench PASS	80443 Marks
	vs. 100000 Marks (i7-6700 @ 3.40GHz)

```
Exit (0)  
wi@wj-virtual-machine:~/ics2018/nexus-am/apps/microbench$
```

图 3.4 native MicroBench 跑分结果

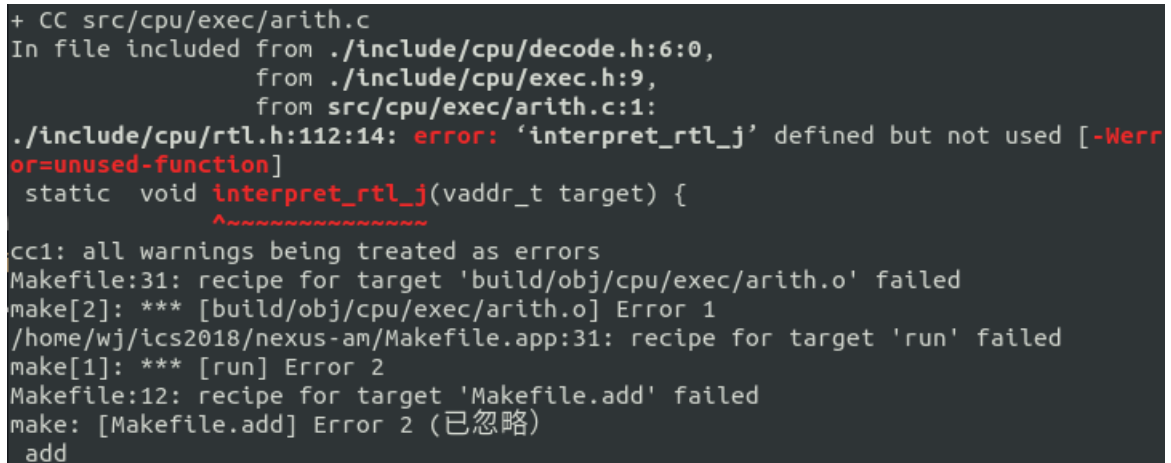
3.2 必答题

3.2.1 编译与链接

- 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 3.5, 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。



```
+ CC src/cpu/exec/arith.c
In file included from ./include/cpu/decode.h:6:0,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/exec/arith.c:1:
./include/cpu/rtl.h:112:14: error: 'interpret_rtl_j' defined but not used [-Werror=unused-function]
static void interpret_rtl_j(vaddr_t target) {
cc1: all warnings being treated as errors
Makefile:31: recipe for target 'build/obj/cpu/exec/arith.o' failed
make[2]: *** [build/obj/cpu/exec/arith.o] Error 1
/home/wj/ics2018/nexus-am/Makefile.app:31: recipe for target 'run' failed
make[1]: *** [run] Error 2
Makefile:12: recipe for target 'Makefile.add' failed
make: [Makefile.add] Error 2 (已忽略)
add
```

图 3.5 去掉 `static` 编译结果

3. 去掉 `static` 和 `inline`, 编译结果如图 3.6 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 3.6 去掉 static 和 inline 编译结果

3.2.2 编译与链接

• 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?
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.
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.)

1. 答: 31 个。

```

wj@wj-virtual-machine:~/ics2018/nemu$ readelf -s build/nemu | grep dummy
45: 00000000002a6904      4 OBJECT LOCAL DEFAULT 25 dummy
53: 00000000002a690c      4 OBJECT LOCAL DEFAULT 25 dummy
65: 00000000002a6938      4 OBJECT LOCAL DEFAULT 25 dummy
72: 00000000002a693c      4 OBJECT LOCAL DEFAULT 25 dummy
80: 0000000000008110      0 FUNC LOCAL DEFAULT 14 frame_dummy
81: 0000000000211a70      0 OBJECT LOCAL DEFAULT 19 __frame_dummy_init_ar
ray_
86: 00000000002156f8      4 OBJECT LOCAL DEFAULT 25 dummy
89: 0000000000215708      4 OBJECT LOCAL DEFAULT 25 dummy
99: 0000000000216748      4 OBJECT LOCAL DEFAULT 25 dummy
108: 00000000002167a8      4 OBJECT LOCAL DEFAULT 25 dummy
114: 00000000002167d0      4 OBJECT LOCAL DEFAULT 25 dummy
123: 0000000000296840      4 OBJECT LOCAL DEFAULT 25 dummy
126: 00000000002a6908      4 OBJECT LOCAL DEFAULT 25 dummy
135: 00000000002a6910      4 OBJECT LOCAL DEFAULT 25 dummy

```

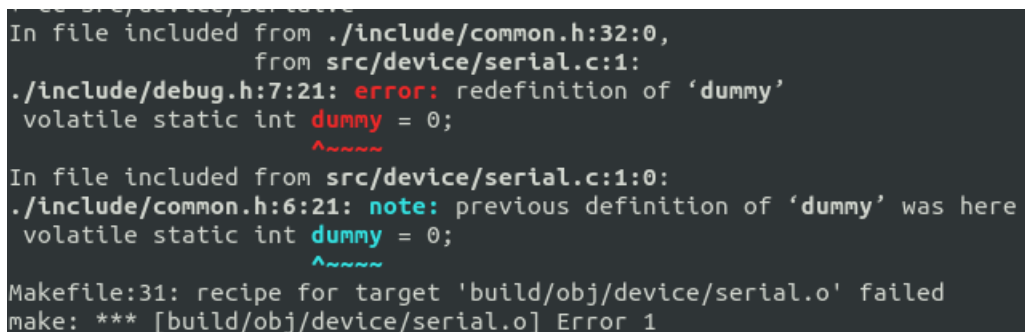
图 3.7 符号表中的 dummy

使用命令 “`readelf -s build/nemu | grep dummy`” 在 `nemu` 的符号表中查询 `dummy` 符号, 结果如图 3.7 所示。再使用命令 “`readelf -s build/nemu | grep dummy | wc -l`” 统计

dummy 出现的次数。结果为 31。

2. 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. 答：编译结果如图 3.8 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。



```
cc src/device/serial.c
In file included from ./include/common.h:32:0,
                 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:6:21: note: previous definition of 'dummy' was here
volatile static int dummy = 0;
                   ^~~~~
Makefile:31: recipe for target 'build/obj/device/serial.o' failed
make: *** [build/obj/device/serial.o] Error 1
```

图 3.8 dummy 初始化后编译结果

3.2.3 了解 Makefile

- 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.) 关于 `Makefile` 工作方式的提示:
 - `Makefile` 中使用了变量, 包含文件等特性
 - `Makefile` 运用并重写了一些implicit rules
 - 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
 - RTFM

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，而不是立即数本身，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。

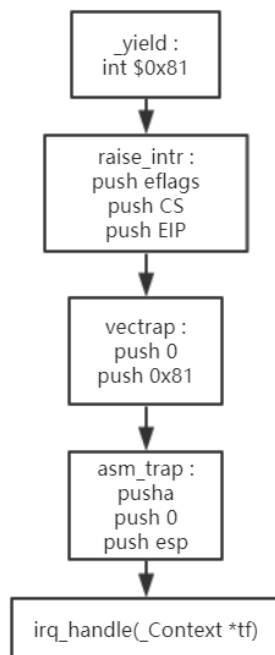


图 4.1 中断处理流程

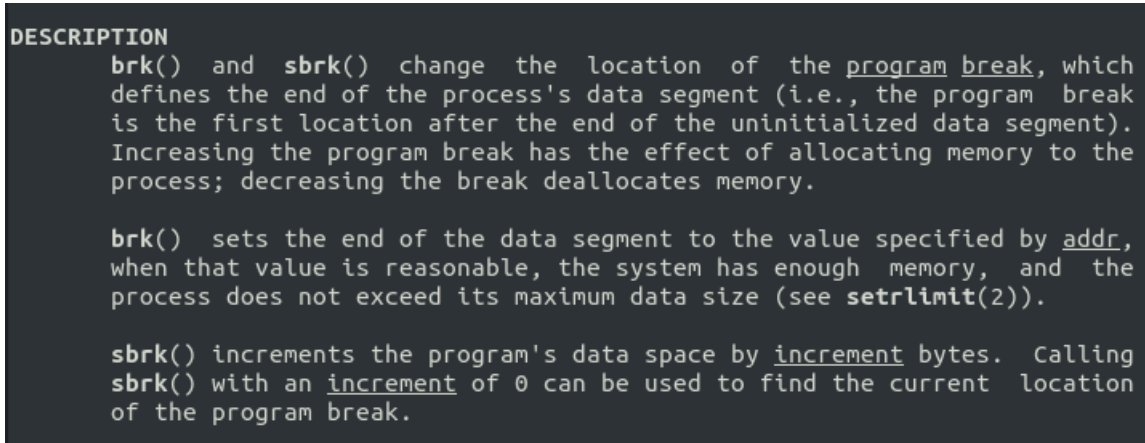
最后入栈的 `esp` 为 `irq_handle` 的参数 `_Context *tf`，即上下文结构体的指针，故其之上入栈的内容为上下文 `_Context` 结构体的内容。综上，上下文结构体的内容依次为保护地址空间指针、8 个通用寄存器（逆序）、中断号、错误代码、`eip`、`cs`、`eflags`。

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。查阅 man 关于 `sbrk()` 和 `brk()` 的行为，结果如图 4.2。 `brk` 通过参数 `addr` 设置 `program_break`，`sbrk` 通过参数增加 `sbrk`，如果增量为 0，可以获取当前 `program_break`。



```
DESCRIPTION
brk() and sbrk() change the location of the program break, which
defines the end of the process's data segment (i.e., the program break
is the first location after the end of the uninitialized data segment).
Increasing the program break has the effect of allocating memory to the
process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by addr,
when that value is reasonable, the system has enough memory, and the
process does not exceed its maximum data size (see setrlimit(2)).

sbrk() increments the program's data space by increment bytes. Calling
sbrk() with an increment of 0 can be used to find the current location
of the program break.
```

图 4.2 `brk()` 和 `sbrk()` 行为

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。设置相应的系统调用。

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.c` 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，提示有指令未实现，如图 4.3 所示。

发现问题：在 `/navy-apps/apps/pal/build` 中生成了 `pal` 的可执行文件。使用 `objdump` 获取反汇编代码，查找出该条未实现的指令，如图 4.4 所示，为 “`stos`” 指令。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。由于 NEMU 中没有涉及到分段，故 ES 段寄存器当 0 处理。

```
PAL_InitGlobals success
PAL_InitFont success
PAL_InitUI success
PAL_InitText success
invalid opcode(eip = 0x04000ce2): ab c7 05 dc cd 15 04 04 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x04000ce2 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x04000ce2) in the disassembling result to distinguish which case it is.

If it is the first case, see

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

nemu: ABORT at eip = 0x04000cea
```

图 4.3 pal 运行失败，指令未实现

```
4000cdc: 81 c7 05 dc cd 15 04      add    $0x413cdc,%edi
4000ce2: ab                          stos   %eax,%es:(%edi)
4000ce3: c7 05 dc cd 15 04 04      movl   $0x4115cd,%edi
```

图 4.4 未实现的 stos 指令

5 PA4 分时多任务

5.1 功能实现的要点

5.1.1 阶段 1——多道程序

本阶段任务：实现上下文切换；实现多道程序系统。

该阶段任务较简单，按实验任务书完成即可。

5.1.2 阶段 2——虚存管理

本阶段任务：在 NEMU 中实现分页机制；在分页机制上运行用户进程；在分页机制上运行仙剑奇侠传；支持虚存管理的多道程序等。

完成本阶段，首先需要阅读相关资料理解 i386 的分页机制，其次，理清 Nanos-lite 是如何将程序加载进内存的，以及内核空间与用户空间的区分。为帮助理解，可以在程序中打印一些关键的地址，来推测 Nanos-lite 的加载机制。通过该方法，我获知了内核空间 CR3 的地址、程序加载后的初始 `program_break` 位置（紧接在加载程序后）等信息，随后也就知道了该如何编写代码完成任务。

对于指导书上所说的最难的一道选做“支持开机菜单程序的运行”。思路是加载一个新进程，覆盖当前菜单进程的 PCB。通过开机菜单 `Init` 的代码可知，切换进程是通过系统调用 `SYS_execve` 实现的。原本该调用是通过 `naive_load` 进行加载新进程，实现分页机制后，此方式不再可行，而应通过 `current_uload` 进行加载，并以当前 PCB“`current`”作为参数传递，如果想以新启动一个进程而不关闭菜单的话，也可以以一个新 PCB 指针作为参数传递。加载完成后，还需要将新进程的上下文作为返回值从系统调用函数返回，使得 `iret` 能正确“恢复”上下文到新进程。由于上下文切换实际上是通过修改 `esp` 及一系列 `pop` 操作来替换寄存器的值，为了不破坏堆栈结构，造成未知的后果，返回上下文指针时，应将新进程的上下文结构体拷贝到传进参数指针所指的位置。做此任务时，也可以通过打印相关信息，来获知哪个地方出现了问题。

5.1.3 阶段 3——分时多任务

本阶段任务：实现抢占多任务；展示你的计算机系统。

使用按键 F1、F2、F3 可以切换前台程序，在 `events_read` 函数中判断是否是这三个按键的事件，进而修改变量以控制前台程序的 PCB。

5.2 必答题

5.2.1 分时多任务的具体过程

分时多任务的具体过程 请结合代码，解释分页机制和硬件中断是如何支撑仙剑奇侠传和hello程序在我们的计算机系统(Nanos-lite, AM, NEMU)中分时运行的。

答：AM：系统先在 `_vme_init` 函数中对内核空间进行分页，并将 CR3 设置成页目录地址。`_protect` 函数完成对用户内存空间的初始化，拷贝内核空间的页目录。`_map` 函数完成虚拟地址到物理地址的映射。`_switch` 函数切换 CR3 内容。

Nanos-lite：通过 `loader` 函数将用户进程通过按页的方式加载到 0x8048000 的虚拟地址，0x8048000 的首地址保证了地址映射时不会与内核空间重叠。`new_page` 函数分配新的页。`mm_brk` 函数在操作系统层维护用户进程的堆空间。通过 `_yield` 来触发 `schedule` 函数实现进程切换。时间中断 `_EVENT_IRQ_TIMER` 会触发 `_yield` 切换进程实现分时多任务。通过上下文管理的一套机制实现上下文切换。

NEMU：`vaddr_read` 函数和 `vaddr_write` 函数完成对虚拟地址的读写，它们通过 `page_translate` 函数进行地址的翻译。

整体流程大致为，Nanos-lite 首先通过 `am` 初始化内核空间，将若干个进程以页的方式加载到各自的内存用户空间。Nemu 实现虚拟地址到物理地址的翻译与读写工作。分页机制实现了各进程可以按自己的虚拟地址进行内存访问，而不用担心实际物理地址是否会与其他进程冲突。硬件中断实现了进程之间的分时多任务调度。当时钟中断到时，触发 `_yield` 函数，再触发 `yield` 中断，调用 `schedule` 函数完成进程切换。

5.3 主要故障与调试

故障 1: 阶段 2 一开始测试分页机制时, 运行仙剑奇侠传会触发错误。

调试发现问题: 一开始找了很久未能找出 bug, 后来开启 DiffTest 跑了将近十分钟, DiffTest 报出了一条与 qemu 不同的信息。查看汇编代码, 发现上一行指令为 mov 立即数。正确的值应该是 0x4, 但我的 nemu 得到的为 0x4000000。但由于前期经过大量测试, mov 指令使用的非常频繁, 理应不会出错。后来发现该条指令立即数长度为 4, 地址跨越了页, 所以发现问题出现在 vaddr_read 函数中处理读取跨虚拟页边界的地方, 将内容顺序读反了, 数据的低位应该在低地址。

解决问题: 修复上述 bug。并改进了读取方式, 原本循环 length 次, 每次读取一个字节。改进为读取两次, 一次读完当前页剩余的字节, 第二次读取剩余字节所在页的剩余字节。

故障 2: 实现 mm_brk 时, page_translate 函数中触发 assert, present 位为 0。

调试发现问题: 通过 print 输出, 当前 brk 和需要的新 brk, 一开始进程需求大量的堆区。意识到没有给 PCB 的 brk 赋初始值。为了获取初始值, 发现程序第一次请求的 brk 值正好等于 DEFAULT_ENTRY 加上文件的大小, 程序通常第一次会通过_sbrk(0)来获取当前的 program_break, 故得知了初始值, 即紧跟在进程数据之后。

解决问题: 修复上述问题, 在 load 函数中, 将程序数据按页读取后, 将 DEFAULT_ENTRY 加上程序数据大小后的值作为进程的初始 program_break。

故障 3: 实现支持开机菜单时, 只执行开机菜单一项程序, 功能正常。但运行多项程序, 开机菜单切换应用后, 很快发生错误, 且有时错误是段错误, 有时是地址翻译触发 Assert, 有时是未实现的指令等。

调试发现问题: 根据上述错误发生的条件及结果, 可初步判断是程序读取的地址错误或者无法正确翻译, 且与多道程序有关。并且程序是在菜单切换运行了一会后才崩溃, 故应该是程序调度后发生了问题。于是反应过来, 我在实现支持开机菜单, 加载新进程时, 误以为要将系统恢复成初态, 故将空闲内存页指针重置为初值并将已分配出去的内存页清空。因此, 导致了切换到其他进程时, 页数据被清空, 进而发生错误。

解决方法: 删去对内存页指针与内存页的重置操作, 以及附带的额外操作。如此后, 发

现实现支持开机菜单并没有想象中那么复杂。

目前该系统有一个缺陷，便是没有对内存页回收的动作，这样内存终有耗尽的时候。

6 总结与体会

经过 3 周的时间，我完成了 PA0~PA4 的所有必做任务以及部分选做任务。最终完成了一个 x86 子集——n86 模拟器 (NEMU)，并在 NEMU 上运行一个简单的小型操作系统，并分时多任务地运行包括仙剑奇侠传的多个进程。

这项实验作为本科阶段最后一项实验，很好的梳理与巩固了之前学过的知识，包括操作系统、组成原理、汇编语言、编译原理等。整个实验完成下来虽然说代码量不大，但需要对计算机组成原理与操作系统基础原理有较深的理解，还需要阅读大量代码，查阅 i386 手册以及 man 手册。

ics 实验设计的非常好，挑战性与趣味性兼具。在实验中不仅让学生巩固了理论知识，学以致用，并且注重于培养 CS 学生的能力与素养。

从 PA1 到 PA4，可以说难度逐渐增大，其中很大的一个原因是指导书给的提示越来越少了。从 PA3 开始，如果不仔细阅读代码，理解代码间的联系与执行流程，做实验时会感到无从下手。但多读几遍指导书，以及 RTFSC 后，找到了线索后便柳暗花明。随着一次次的“柳暗花明”，我对计算机组成原理和操作系统有了更深的理解。如 PA2 中，再次熟悉了计算机是如何取指、译码、执行的，PA3 和 PA4 则了解了操作系统的基础工作原理。

完成这个小型虚拟计算机，我体会到“虚拟”或者“抽象”这一概念，在计算机中随处可见。例如 AM 抽象机，“一切皆文件”将设备也抽象成文件。“抽象”的好处使计算机分层更容易，不同的层次不需要关心其他层次具体的细节，大大降低了计算机实现的难度，增加了通用性。如此，开发应用程序的程序员不需要关心机器底层的架构，不需要为不同的机器适配不同的版本，只需要使用统一的接口编程即可。

当然，除了巩固计算机理论知识外，我认为这实验对于学生的能力培养也非常重要，这项实验的另一个目标，就是将大家培养成一个素质合格的 CSer，具备独立解决问题的能力。正如实验指导书上所说，“如果你只是仅仅把 PA 作为一个编程大任务，我们相信你确实吃了亏”。这项实验锻炼了我对于 Linux 的使用、查阅英文手册和资料等，让我认识到基础设施的重要性。

不过这次实验时间较短，实验本身又有一定难度，并且大四才做这些培养时间上

也有点晚。很多同学可能不会顾及到这些对于 CS 能力与素养的训练，那确实是浪费了一次大好的锻炼机会。不得不说我们学校相比于南京大学，在这方面稍微欠缺了些。我周围很多同学，在做这次实验前，甚至不会使用 `git`。虽然说能力的训练更应该依靠自己，但学校在教学理论知识外，也应注重对 CS 学生能力与素养的培养。这也是我对我校计算机的**建议**。大二大三期间，有不少实验设计的不太好，时间的投入与实际收获不对等。类似本次实验和组成原理这样的优秀的实验与课设数量较少。当然，这样的实验会需要老师们或者助教们投入大量的时间与心血。但这样的有意义的实验是值得的。如果能在大二时就通过这种能力培养，虽然这个实验本身可能会花费学生大量时间，但学生能力提高后，养成良好的习惯，具备独立解决问题的能力，之后再遇到复杂的项目也不太会感到畏难。

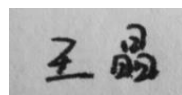
• 指导教师评定意见 •

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：_____
