Programming Assignment 实验报告

2015 年 12 月 16 日

Fudan University

曹景辰

复旦大学

计算机科学系

Phone ： 13524128246

E − Mail： 14307130003@*fudan.edu.cn*

PA 第一阶段实验报告

PA 第二阶段实验报告

PA1.0 引入

思考题：初始虚拟化

“Hello World program”
Micro operating system
Simulated x86 hardware
NEMU
GNU/Linux
Simulated x86 hardware
VM workstation
Windows
Computer hardware

思考题：究竟要执行多久？

当参数输入 - 1 的时候，程序会不停模拟执行下去，直到执行到“GOOD TRAP”，也就是全部模拟完成，才会停下。

思考题：谁来指示程序的结束?

main 函数执行完成后，还会执行一段代码，对全局对象析构，释放对内存的使用权。

PA1.1 基本功能

CPU_state 结构下要求实现模拟 x86 寄存器的结构，提示告诉我使用匿名 union。于是我将原来的 struct 改成了 union，其中有两个部分。第一部分是 gpr 数组，以规定好的顺序存放了诸多寄存器，保持原样不作改动。第二部分中，我将每个 32 位寄存器，以及它对应的一个 16 位寄存器两个 8 位寄存器放在一个 union 里，再将这么多 union 构成的 struct 作为第二部分，和之前的 gpr 数组放在一个 union 中。这样重组后的 reg.h 定义的 CPU 结构，既可以用 gpr 数组加下标的方式访问寄存器，也可以直接用“.+ 寄存器名字”直接访问。节选的关键代码如下

```
union
{
        uint32_t eax;
        uint16_t ax;
        struct
        {
                uint8_t al,ah;
        };
};
```

RTFSC 之后，我发现框架代码用 readline 函数读入一行输入，用 strtok 命令，将输入的字符串从第一个空格处断开，前半段匹配各种 NEMU 命令，后半段作为参数调用对应的函数。

si 命令很显然，直接用适当参数调用 cpu_exec 就可以。参数注意从字符串转成 int 即可，没有参数默认为 1。

不得不说 sscanf 和 sprintf 很好用，处理一些繁琐的数据类型转换时候非常方便，虽然效率比起手动写代码类型转换不高，但是省下了大量的时间。

info r 如法炮制，参考 GBD 的格式后，按个 printf 寄存器的值就可以了。

扫描内存命令比起前几个稍稍麻烦了一些，输入的参数分段识别为一个 int 和一个表达式后，表达式部分用后文完成的表达式求值函数求出对应的 unsigned int 值，把它们作为参数调用 swaddr_read 函数，依次输出各个值。(参考了 GDB 的输出格式)

PA1.2 表达式求值

第一阶段最花时间的一个部分。我花了不少时间阅读了正则表达式的规则，功能实在太强大。以前总觉得识别 16 进制数十分麻烦，要先匹配“0x”，在匹配后面几位，检查是不是合法。用了正则之后，判别规则只要“0x[a-fA-F0-9]+”短短十几个字符就完成了，佩服的五体投地。

注意：1)、正则元字符识别时候要加上“\”，但是 C 语言中“\”也是一个元字符，所以要用“\\”。

2)、为了防止!= 被识别成! 和 = ，规则数组中，!= 对应的规则要在! 和 = 之前。其它同理。

我的做法是，编写好各种识别规则后，用正则表达式将对应的 token 放入数组。我修改了原来的 token 结构，加入了 num 成员，使得我识别出一个 token 为十进制数或十六进制数或寄存器的时候，直接将对应的值存入，免去了以后再对 str 字符串处理的过程。随后，我放弃了参考手册中建议的用递归方法求表达式的思路，选择了时间复杂度更低的后缀表达式求值。不仅免去了递归调用中可能出现的溢出错误，也大大加速了计算速度 (虽然递归也不用花很多时间)。转后缀表达的时候用到了一个运算优先级表，好在没有双目右结合运算符，程序也不复杂。再对后缀表达式模拟一个栈运算出最终结果即可。

除了?:、->、. 别的运算符我基本都实现了。

思考题：实现带有负数的算术表达式求值

已经完成，具体方式直接看源代码好了。我对负号做了一个特判，区分了减法和负号，再执行相对应的运算。当然啦，因为返回值要求是 unsigned int，所以小于零的结果会以一个很大的正数的形式出现。

PA1.3 观测点

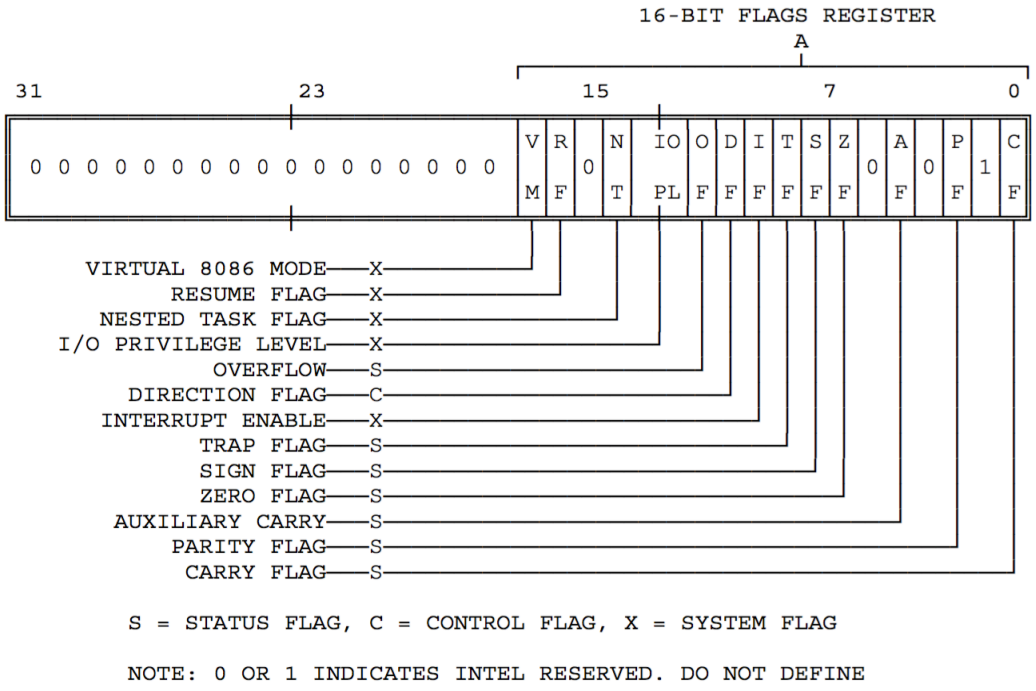
感觉没有什么可说的，完全的链表模拟。

链表每次都从表头上插入或取出，可以节约一个指针变量。记得注意要在头文件中加入新写的函数声明，并在 cpu_exec 中加入几个缺少的头文件。

必答题

问：EFLAGS 寄存器中的 CF 位是什么意思?

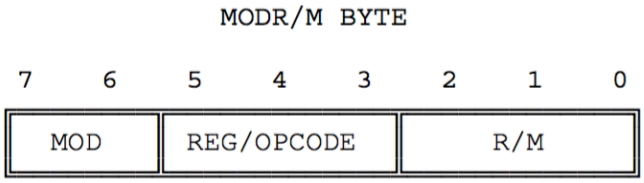
答：参阅 i386 手册 Figure 2-8 处。CF 是进 / 借位标志，用来表示无符号数加减运算时的进 / 借位。加法时，若 CF = 1 表示加法有进位；减法时，若 CF = 1 表示不够减，因此，加法时 CF 就应该等于进位输出 C，减法时，应将进位输出 C 取反来作为借位标志。



问：ModR/M 字节是什么?

答：参阅 i386 手册 17.2.1。ModR/M 字节出现在许多 80386 指令中，它由三部分组成。Mod 域有两位，它和 r/m 一同决定了 32 种不同的可能值。可能值由 8 个寄存器和 24 个索引组成。reg 域占 3 位，它的意义由 opcode 决定，可以表示 8 个寄存器中的一个，或是 opcode 代码的补充。r/m 域占 3 位，它要和之前的 Mod 域结合起来分析表明对应的寄存器。

Figure 17-2. ModR/M and SIB Byte Formats



问：mov 指令的具体格式是怎么样的?

答：i386 手册 17.2.2.11。具体格式见截图。

MOV — Move Data

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

MOV — Move to/from Special Registers

Opcode	Instruction	Clocks	Description
0F 20 /r	MOV r32,CR0/CR2/CR3	6	Move (control register) to (register)
0F 22 /r	MOV CR0/CR2/CR3,r32	10/4/5	Move (register) to (control register)
0F 21 /r	MOV r32,DR0 -- 3	22	Move (debug register) to (register)
0F 21 /r	MOV r32,DR6/DR7	14	Move (debug register) to (register)
0F 23 /r	MOV DR0 -- 3,r32	22	Move (register) to (debug register)
0F 23 /r	MOV DR6/DR7,r32	16	Move (register) to (debug register)
0F 24 /r	MOV r32,TR6/TR7	12	Move (test register) to (register)
0F 26 /r	MOV TR6/TR7,r32	12	Move (register) to (test register)

问：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?

答：从 terminal 进入 nemu 后，输入

```
find . -name "*.[ch]"|xargs cat|wc -l
```

就会显示当前的代码总行数。然后输入 gitk，找出最早的版本的 hash 号，git checkout 到最早的版本，再次输入

```
find . -name "*.[ch]"|xargs cat|wc -l
```

将两次得到的结果相减即可。截图如下，得出结果——787

```
[caojingchendeMacBook-Pro:nemu caojingchen$ find . -name "*.[ch]"|xargs cat|wc -l]
    3698
[caojingchendeMacBook-Pro:nemu caojingchen$ git checkout master                  ]
Previous HEAD position was 57510cf... >  compile NEMU 141220000 cjc Linux ubuntu
 3.19.0-25-generic #26~14.04.1-Ubuntu SMP Fri Jul 24 21:16:20 UTC 2015 x86_64 x8
6_64 x86_64 GNU/Linux  16:12:45 up 2 min,  2 users,  load average: 0.56, 0.33, 0
.14 5ff7b35b149819f98fde52aa907a50c5edb6d5
Switched to branch 'master'
[caojingchendeMacBook-Pro:nemu caojingchen$ find . -name "*.[ch]"|xargs cat|wc -l]
    4485
```

问：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：输入

```
find . -name "*.[ch]"|xargs cat|grep -v ^$|wc -l
```

就是去空行版的。截图如下，得出结果——773

```
149819f98fde52aa907a50c5edb6d5
[caojingchendeMacBook-Pro:nemu caojingchen$ find . -name "*.[ch]"|xargs cat|grep ]
-v ^$|wc -l
    2945
[caojingchendeMacBook-Pro:nemu caojingchen$ git checkout master                  ]
Previous HEAD position was 57510cf... >  compile NEMU 141220000 cjc Linux ubuntu
 3.19.0-25-generic #26~14.04.1-Ubuntu SMP Fri Jul 24 21:16:20 UTC 2015 x86_64 x8
6_64 x86_64 GNU/Linux  16:12:45 up 2 min,  2 users,  load average: 0.56, 0.33, 0
.14 5ff7b35b149819f98fde52aa907a50c5edb6d5
Switched to branch 'master'
[caojingchendeMacBook-Pro:nemu caojingchen$ find . -name "*.[ch]"|xargs cat|grep ]
-v ^$|wc -l
    3718
```

问：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 是开启所有 warning 选项， - Werror 是将所有 warning 变成 error。使用这两个命令，可以强迫以前总是无视 warning 的程序员们注意这些 warning，从而规避一些难以发现的 bug。如，int 类型赋值给 char 类型时，可能会出现位溢出错误，如果不用这两条命令，很有可能就被忽略了，导致熬夜改 bug 但依旧不知原因的惨剧。

PA2.1 运行 mov-c 程序

这个阶段的难点在于读懂框架代码。具体的框架代码如何运行，在此我就不多言了，虽说手册中写的挺详细的，但是完全理解也花了我不少时间。

遇到的一个困难是 eflags 的实现。x86 系统中，要改变 eflags 的指令非常多，好在大部分比较简单，如：test，cmp 等命令，对 eflags 的操作也相对简洁。但是，加减运算的 eflags 操作不仅繁多，而且针对不同的指令，因为操作数的不同，要分别编写不用的 eflags 处理程序比较麻烦，所以，我选用了数字逻辑电路课上讲过的 ALU 来处理。根据不同指令，设置四种对应的控制端输入，用 C 语言模拟 ALU 电路的运算过程，随后把这段程序写成宏定义放在 instrument_all.h 中，极大地简化了程序编写的复杂度，而且相比自己按个写 eflags 操作更不容易出错。

思考题：不能返回的 main 函数

根据 GDB 的调试，main 函数结束后，会执行一系列过程。大致内容是释放程序申请的内存，全局变量的析构等等。但是 nemu 没有这些过程，所以 main 函数 return 之后会回到 init 过程中，call main 的下一句，通常是我自己编写的某个函数，随后在执行一次。但是因为各个寄存器中没有正确的数值，所以程序的运行过程不可预知，会执行到各种奇怪的命令。因为我 nemu 的完成度较低，所以会因为没有完成所有指令而跳出错误。

PA2.2 简易调试器（2）

这个阶段要求实现全局变量的读取和打印栈帧链。

要实现读取全局变量，要先读取文件的符号表信息，这部分框架代码已经实现了。在 terminal 中 man elf 之后，发现 Elf32_Sym 这个 struct 的结构形式。从 st_name 获得某个符号的名字在字符串表中的偏移量，核对后，返回这个变量名对应的地址偏移量，再按照 4 个字节输出就可以了。

打印栈帧链需要读取 ebp 寄存器的内容。ebp 在内存中的存储类似链表，可以读取 ebp 的内容找到每个栈上一层对应的栈帧地址。函数调用时的参数，存储在 ebp 高 8 个字节的地址以上的空间中，依次以 4 个字节形式输出 4 个输出即可。为了获得函数名，ebp 高 4 个字节的地方是函数的返回地址，用和之前叙述的相同方法获得对应的函数名即可。

思考题：消失的符号

gcc 编译的程序，默认不会记录局部变量名，局部变量不是一个符号，所以符号表中也不会记录。所谓符号是静态变量，全局变量和函数名。

思考题：寻找“Hello world!”

“Hello world”不是一个符号，所以不再符号表里。这是一个字符串常量，所以存放在.rodata节中，不在字符串表里。

思考题：丢失的信息

我发现输出的内容是一串奇怪的数字。因为字符串中每一位内容都是 ascii 码形式储存的，调试器会把 str[1] 对应地址的内容，按照 4 字节的形式输出。这个问题目前无法解决，因为符号表没有储存每个符号对应的变量类型。这反应了编译器在编译链接重定位过程中，丢弃了不少源代码信息。

思考题：冗余的符号表

可执行文件丢弃符号表信息后可以运行，没有任何影响。但是可重定位文件丢弃符号表信息后就不能链接了。因为文件在链接过程中，要根据符号表，把来自于其他文件的信息填入对应的位置。没有符号表后编译器不知道该把哪些内容填入，所以会出问题。但是可执行文件可以视为一堆机器码的集合，机器执行的必要内容都在里面了，哪怕没有符号表也可以正常运行。

思考题：ebp 是必须的吗？

开启-O2 优化后，GDB 依旧可以打印栈帧链，由此可见 ebp 不是必须的。具体怎么完成的。。。。。。。我实在看不懂-O2 优化后的汇编代码，网上也没查到相关内容，抱歉无法回答。