PA2-1 report

马逸君 17300180070

本次实验我完成了所有内容（添加指令、模拟断点、变量支持、打印栈帧链）。

添加指令

俗话说，最令人恐惧的是恐惧本身。俗话又说，万事开头难。当我还没弄清楚我们这次实验要做什么的时候，我的内心是崩溃的。我阅读了两遍讲义，然后又速读了一遍框架代码，其实一开始我是**基本没有看懂那些宏定义**的（展开以后也看不懂），但我大致知道我们需要做什么了，所以就采取了照猫画虎的办法。

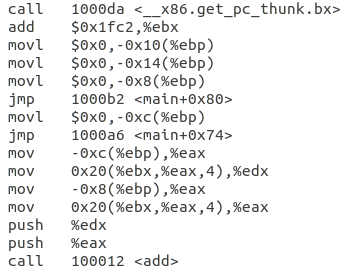
当然仅仅照猫画虎是不够的，我们需要大致弄清楚框架代码的结构，才能正确通过编译。框架代码中的一个指令需要在五个文件中写相关代码，它们分别是（"\*"表示指令名称）：  
all-instr.h，exec.c，\*.c, \*.h, \*-template.h。写完add指令后，出于显而易见的原因，我通过直接**复制**add源码**并修改**的方式来写sub源码——在一个充斥着宏定义的工程里，这其实**是非常愚蠢的行为**，你永远不知道哪里会出现莫名其妙的行为，比如我把add-template.h拷贝成sub-template.h以后，仅仅是忘记把里面的#define instr add改成#define instr sub，就导致编译器报了一堆看不懂的错，因为看不懂，我只能回去把源码重新检查了一遍，才发现问题所在。

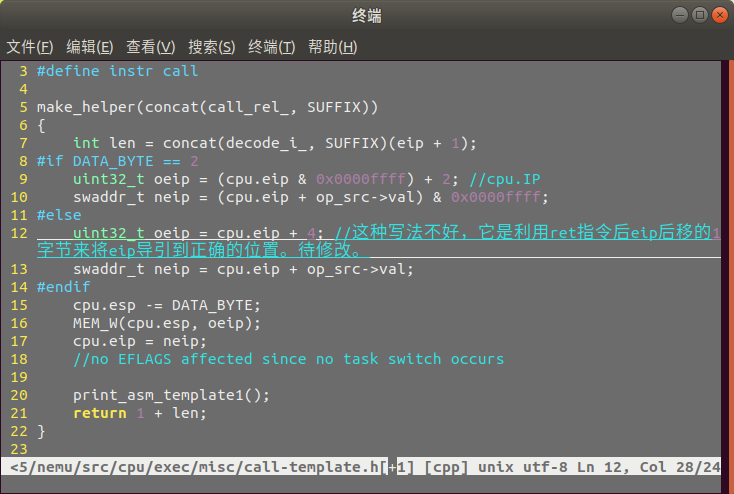
之后在更新EFLAGS的时候，为了让编写出来的更新语句能在所有指令里使用，我就把它们写到了template-start.h里面。在写这个文件的时候我注意到一个现象：所有在template-start.h里定义的宏，最后全部都在template-end.h里解除定义(undef)了。这一设计也很好理解：因为#include语句仅仅是简单的文件内容替换，在头文件里定义的宏也会在包含它的用户源代码里起作用，而这并不一定是我们所希望发生的，还可能引起各种很难发现的问题，所以在头文件里用到的宏一定要在头文件末尾处undef掉。所以说要养成良好的**宏的使用习惯**。

在编写Jcc指令的时候，i386手册对它的功能是这么描述的：EIP := EIP + SignExtend(rel8/16/32); 受到之前编写的call、jmp等一些指令的影响（在这些指令中，立即数只有16位和32位两种情况，只需要给EIP直接加上立即数就可以了：EIP := EIP + rel16/32;），我就直接写成了cpu.eip += op\_src->val，结果可想而知。当我看见屏幕上大大的红字写着"i386 Manual"的时候，我的内心又是崩溃的。之后我通过输出调试的方式，成功定位了问题所在，修复了这一bug。身为一个计算机专业的资深本科生，却**对于符号扩展这一概念毫无敏感性**，真是不应该。

这一部分的最后一个困难发生在**对代码的理解**上。首先是模板代码里的print\_asm()语句，因为一开始我并不知道这行语句是做什么的，所以每次添加新指令的时候，对于拷贝过来的这条语句都是原封不动地留下来的；直到我在运行nemu时执行了si指令，发现在执行到ret语句的时候打印出的汇编语句仍然是上一条指令的汇编语句而不是ret，我花了不少时间RTFSC，才确定这个功能正是print\_asm()语句实现的，然后我又花了更多的时间把已经添加进去的所有指令都改正过来。“Every line of untested code is wrong”；“随着时间推移，发现和修正一个错误的代价会越来越大。”

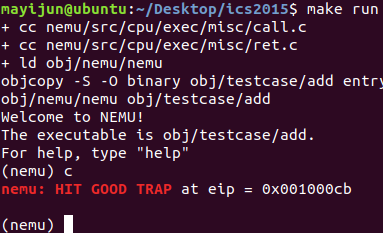
然后是执行指令时return的值的问题，按照定义，return的值是指令的长度，但是实际执行过程中，add.c这个文件却总是hit bad trap。我单步调试并且不停打印寄存器的值，发现基址寄存器%ebp出现了很不寻常的一个奇数：1048637。按理来说，程序里并没有出现过char类型变量，基址寄存器不应该涉及到奇数。

根据调试，问题应该出现在这一段里：

我调了半天不能确定问题所在，甚至还去查了这个\_\_x86.get\_pc\_thunk.bx函数的定义，无济于事。这时我想到call指令里有一处问题：

（图中的注释是我第一次写的时候自己写上去的）

这时我突然顿悟，指令return的值虽然理论上是指令的长度，但它的用处却是导引eip后移到下一指令的正确位置。如果ret指令返回值是1，那么eip在返回以后还会后移一格，为了使eip能指向正确的位置，call指令里压栈的返回地址就必须相应地减去1，这是不好的，因为这样编写出的call指令没有独立性（必须和ret配合使用才能保证正确性）。我尝试了将ret指令的返回值改为0，然后将call指令里的那个"4"改成"5"，再次运行nemu：



这个故事告诉我们两个道理：作为一个成熟的系统程序员，我们**必须对代码理解到位**，必须做到对写下的每一行代码都心里有数，要清楚它们的含义、它们的作用都是什么；在写程序的时候，良好的注释习惯可以为后续开发调试节省下可观的时间成本和机会成本。

思考题：

1. 对于以下5个集合:
   1. 所有 instruction prefix
   2. 所有 address-size prefix
   3. 所有 operand-size prefix
   4. 所有 segment override prefix
   5. 所有 opcode 的第一个字节

它们是两两不相交的. 这是必须的吗? 这背后反映了怎样的隐情?

答：是的，否则一条指令可能有多种解释，产生歧义，汇编器将无法在读到指令的第一个字节时唯一确定这个字节的含义，从而无法解释指令。

1. Motorola 68k系列的处理器都是大端架构的,考虑以下两种情况:
   1. 假设我们需要将NEMU运行在Motorola 68k的机器上(把NEMU的源代码编译成Motorola 68k的机器码)
   2. 假设我们需要编写一个新的模拟器NEMU-Motorola-68k, 模拟器本身运行在x86架构中, 但它模拟的是Motorola 68k程序的执行

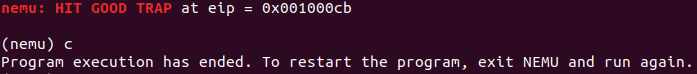
在这两种情况下, 你需要注意些什么问题? 为什么会产生这些问题? 怎么解决它们?

第一种情况：在大端机上模拟大端机。这时不会产生问题，虽然源代码编译成的二进制代码中的常数的顺序会翻转变成大端方式，但是因为instr\_fetch()函数是原封不动地读取的，主机的CPU也会按照大端方式解释它，所以无需修改源码。

第二种情况：在小端机上模拟大端机。这时用户程序编译产生的二进制代码中的常量都是大端方式存储的，主机的CPU则是小端方式，二者是相反的；我们需要修改instr\_fetch()函数使其在读取常量后进行翻转，使其符合主机CPU的数据存储顺序，方可正确运行。

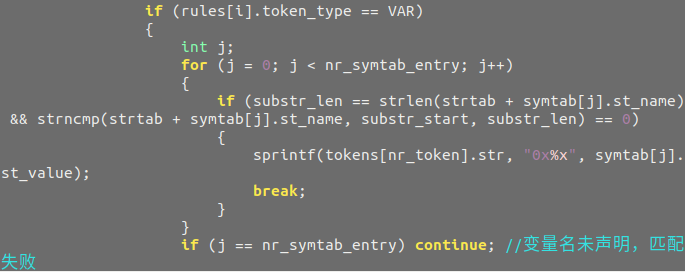
1. 为什么目前让用户程序从 main 函数返回就会发生错误? 这个错误具体是怎么发生的?

答：测试结果显示，如果从main函数返回，eip的值将变为0：

作为对比，正常情况下，在hit good trap之后，nemu的状态如下：

所以，错误发生的原因是，如果让用户程序的main函数返回，nemu无法识别，从而无法将自己的状态从running置为ended，从而在eip变为无效值之后继续试图将eip指向的地址作为指令解释并执行，从而引发错误。

添加变量支持

这一节最大的敌人仍然是“未知”。在我眼中，这一节的代码任务根本无从下手，直到我查阅资料得知，对每一台Linux机器，其ELF文件的结构定义都位于**/usr/include/elf.h**中。我查到了Elf32\_Sym结构的定义，知道了该访问其中的哪些成员，之后就简单了：

开发过程中出现的唯一一个bug出在strncmp()函数身上。一开始我的if条件是这么写的：

strncmp(strtab + symtab[j].st\_name, substr\_start, substr\_len) == 0

这之后产生了一个非常滑稽的现象。因为我记错了变量名，所以在nemu中输入了"p test"，即使这样也能匹配到"test\_data"这个变量。这时我才想起，这种写法将只会比较两个字符串的前substr\_len个字节，substr\_len的意义是输入的nemu命令的参数的长度，它可能小于符号的完整名称！因为参数可能不止一个，所以我们不能直接用strcmp()来将当前这个参数和符号表项作比较，所以不得不在前面"&"上另一个条件：

substr\_len == strlen(strtab + symtab[j].st\_name)

这个问题的解决是非常富有戏剧性的，如果我没有记错变量名，可能根本发现不了这个错误。说到底还是我太菜了，我们要**提高自身硬实力**，考虑清楚每一行程序语句的所有情况和可能出现的各种问题，而不是凭运气调试程序。

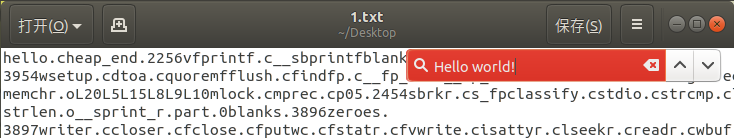
思考题：

1. 我们在 add.c 中定义了宏 NR\_DATA , 同时也在 add() 函数中定义了局部变量 c 和形参 a , b , 但你会发现在符号表中找不到和它们对应的表项, 为什么会这样? 思考一下, 什么才算是一个符号(symbol)?

答：众所周知，“符号”只有三种：在当前模块定义并可以被其他模块引用的全局符号、在其他模块定义并被当前模块引用的外部符号、在当前模块定义且只能被当前模块引用的本地符号。局部变量分配在栈中，是在运行时动态分配的，且不会在过程外被引用，所以不属于符号。符号定义的本质是分配存储空间，对函数名而言指其代码所在区，对变量名而言指其所占的静态存储区，定义的符号的值指的是其目标所在的首地址。宏定义仅仅是简单的文本替换，在程序运行过程中是不存在对应的“实体”（变量或函数）的，不占空间，也没有地址，所以不属于符号。

我认为符号指的是**在重定位过程中完全确定地址的变量或函数**。

1. 在GNU/Linux下编写一个Hello World程序, 编译后通过上述方法找到ELF文件的字符串表, 你发现"Hello World!"字符串在字符串表中的什么位置? 为什么会这样?

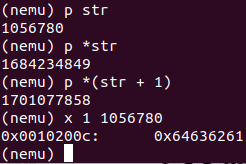
答：不存在。

原因是，字符串常量根本就不是存储在字符串表(.strtab节)中的，而是存储在静态数据区的.rodata节。

1. 在用户程序中定义以下字符数组:

char str[] = "abcdefg";

尝试通过上述方式输出 str[1] 的值, 你发现有什么问题? 运用现有的信息, 你能够解决这个问题吗? 如果能, 请描述解决方法, 并尝试实现; 如果不能, 请解释为什么, 并尝试总结这背后反映的事实.

答：发生的问题是nemu默认按4字节打印数据，所以一次取出来了4个字节的数据，然后当作int型转换成十进制输出。

我认为这个问题理论上能够解决，只要我们能够知道访问的变量的类型，并在代码中作相应修改即可。（但是目前我们还不知道如何得到变量的类型）

1. 在GNU/Linux下编写一个Hello World程序, 然后使用 strip 命令丢弃可执行文件中的符号表:

gcc -o hello hello.c

strip -s hello

用 readelf 查看hello的信息, 你会发现符号表被丢弃了, 此时的hello程序能成功运行吗?

目标文件中也有符号表, 我们同样可以丢弃它:

gcc -c hello.c

strip -s hello.o

用 readelf 查看hello.o的信息, 你会发现符号表被丢弃了. 尝试对hello.o进行链接:

gcc -o hello hello.o

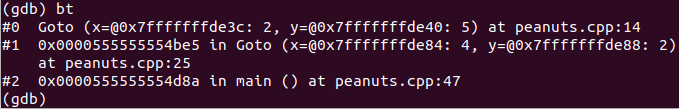
你发现了什么问题? 尝试对比上述两种情况, 并分析其中的原因.

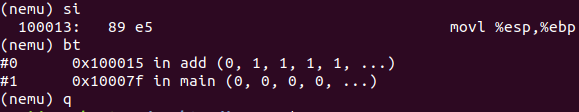
答：对可执行目标文件剥夺符号表后，仍然可以成功运行。原因是字符串常量并不存储在符号表中。

对可重定位目标文件剥夺符号表后，将不能完成链接，报错“对main未定义的引用”。可见错误原因是重定位过程中是需要符号表中的信息的，如果在链接之前剥夺了符号表，将无法确定main的符号定义，从而出错。

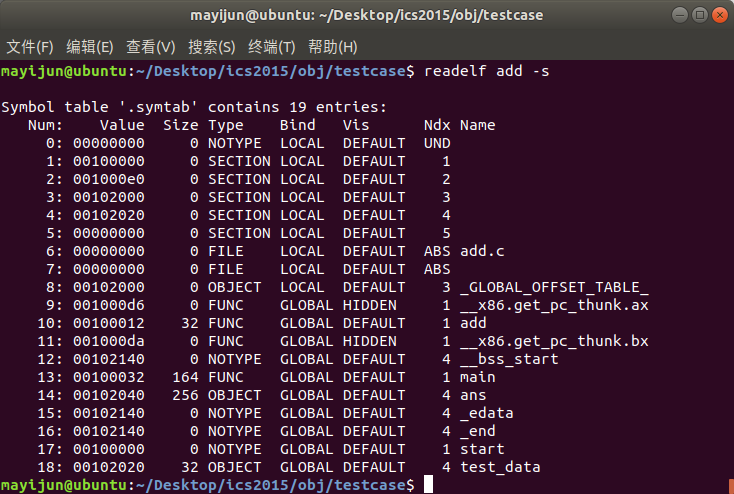
打印栈帧链

打印栈帧链这一节的讲义中有一句话：“如果你从来没有使用过 bt 命令, 请先在GDB中尝试.”很不幸，我就是这个人群的一员。所以在动手编写代码之前，我先拿很久很久以前写的一个递归程序peanuts.cpp尝试了一下gdb的打印栈帧链功能：

（为什么我当时要把参数声明成引用类型？我也不明白）

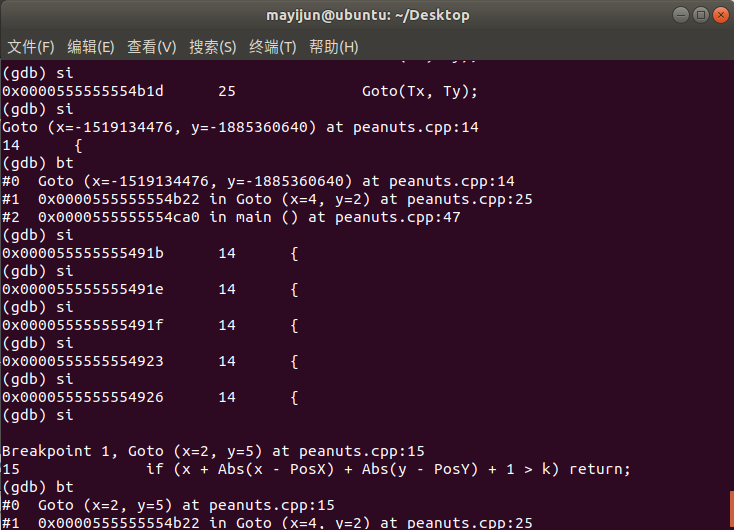
之后的开发过程就相对比较容易了，只遇到了三个难点：第一个是，讲义中要求向高地址强行读出前5个参数，这样就造成了**访问越界**的问题，如果在程序最开始的时候调用bt命令，可能会访问到0x800000以上的非法地址。为了解决这个问题还是花了点时间才调好。

（注意main后面只打印了4个参数，这是防止越界语句的作用）

第二个是，当ebp位于一些名字叫\_start、\_\_x86.get\_pc\_thunk.bx等一些奇奇怪怪的函数里面的时候，我发现我的程序无法输出任何信息。查了符号表才知道，这些函数不知是因为什么奇奇怪怪的原因，它们**在符号表中的size项居然是0**：

这大大超出我的知识水平了，只能在自己的程序中加入对这一情况的特判，输出这样的提示信息：“unknown function（符号表中size项为0的函数，无法识别）”。

第三个是，当我们位于一个函数最开始的三条指令(push %ebp mov %esp,%ebp sub $0x10,%esp)时，**当前函数的栈帧还没有成形**，这个时候打印栈帧链，会看到一些莫名其妙的特效。这个问题还是相当烧脑的，但我后来发现，其实gdb也做不到：

（看到了吗？在当前函数的栈帧成形之前，gdb打印出的参数值竟是x = -1519134476, y = -1885360640，而在成形后方可打印出正确的x = 2, y = 5）

用一句通俗的话来说，这种极端情况“超纲了”。故不予处理。

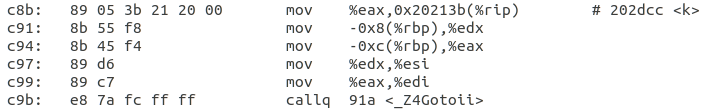
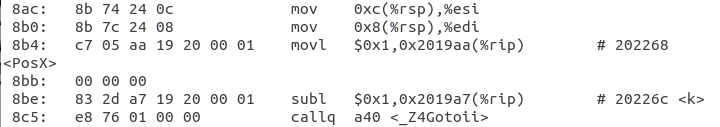
至此，就完成了本次饶有趣味的PA。

思考题：

使用优化选项编译代码的时候, gcc会对代码进行优化, 会将 %ebp 当作普通的寄存器来使用, 不再让其作为指示当前的栈帧, 更多的信息可以查阅 man gcc 中的 -fomit-frame-pointer 选项. 我们使用 -O2 来编译NEMU, 你可以对NEMU进行反汇编, 查看一些函数的代码. 在这种情况下, 代码要怎么找到函数调用的参数和局部变量?

另外优化 %ebp 寄存器之后, 就不能使用上述方法来打印栈帧链了. 如果你使用GDB对NEMU进行调试, 你会发现仍然可以使用bt命令来打印栈帧链. 你知道这是怎么做到的吗? 在优化 %ebp 寄存器之后, 为了打印栈帧链, 还需要哪些信息?

答：因为反汇编nemu这样的大工程有难度，所以我还是使用前面说到的peanuts.cpp进行探究。

对比反汇编代码发现，优化后，访问参数和局部变量的职责由%rsp代替：

至于第二个问题，对比两种版本的返回代码：

因为网上也查不到相关资料，在此我只能说一说自己的猜想：可能是r12、r13、r14、r15中的信息能够辅助定位当前所在的栈帧。