PA1-2 report

马逸君 17300180070

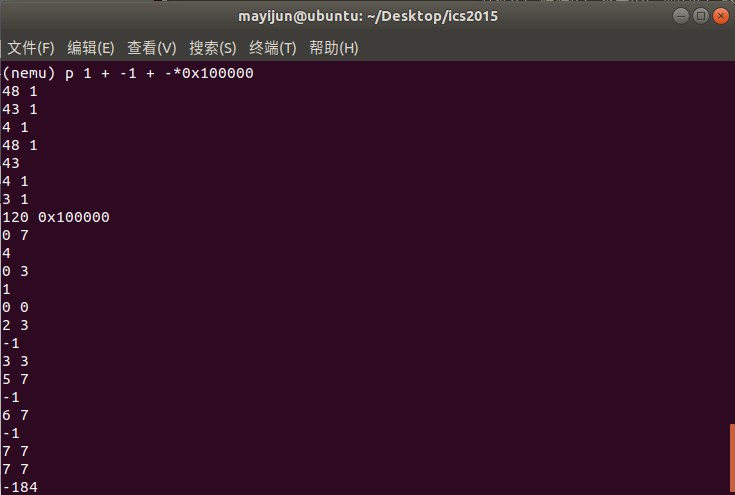
本次实验我完成了所有内容。

最开始的时候，我遇到了一个独特的问题：因为已经一个月没有PA任务了，我忘记了代码结构和git的使用方法。我的解决方法是RTFM，从前几节的讲义中我回忆起了工程的代码结构，从附录中我回想起了git的使用方法。

表达式求值

表达式求值是本次PA的主要难点。在4月30日这天，我花了整整5个小时自学正则表达式基础内容[1]、完全实现词法分析和递归求值。“完全实现”的意思是写出了代码并通过了自己出具测试数据的测试。

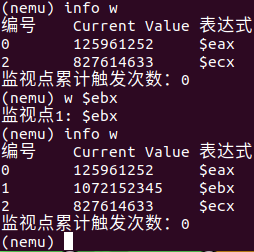
如何出具测试数据是相当困难的：必须考虑有哪些可能的坑，并逐个踩上去。所幸我们在OI时代就接受过相应的训练。我自己出了(1 + 1 + -1 + 1) \* (1 / 1)这样一个算式，再加上讲义里的4 + 3 \* (2 + -1)，测出了一堆问题。

调试也是有技巧的，此处我采取的是输出调试的办法：在调试词法分析时，输出所有token的类型和子串信息；在调试递归求值时，输出表达式每一步求值的下标范围和分裂位置。

完成表达式求值后我重写了cmd\_x()函数。写这个函数的时候碰到了一个问题：如何从args字符串中同时读出一个整数N和一个含空格子串EXPR？我百度得知，scanf()是支持正则表达式的[2]。感谢强大的scanf，问题迎刃而解：

sscanf("%d%[^\n]", &n, exp);

监视点

监视点比较简单，作为从OI时代过来的选手，链表可谓轻车熟路。

需要处理的就是一些小小的格式问题：比如，用info w打印监视点时，如何做到按编号顺序输出监视点信息。我采用的方法是，保持head和free\_这两个链表的有序性，即保持它们的编号始终是随遍历顺序递增的。这只需要在每次new\_wp()链表插入时，把简单的头插改成遍历找到正确的位置插入就可以了。

讲义最后提供了一篇介绍断点实现方式的参考文章。它的大意是，真实的调试器实现断点并不是借助监视点实现的（尽管理论上可以），而是将断点所在地址的内容用int3指令代替。当执行到int3指令时，操作系统内核会发出暂停信号，形式上类似于异常处理。用户可以在此时查看栈和各个变量的内容。当继续运行时，调试器将PC回滚1个字节，将被int3代替的原指令放回，执行完原指令后，再继续用int3代替。

思考题：int3指令的长度必须只能为1字节吗？是的，否则当我们在一些长度为1字节的指令上设置断点时，int3指令会覆盖后面指令的一部分，从而可能产生无效指令。例如参考文章里的这段汇编(伪)代码：

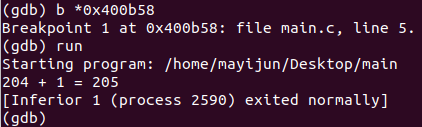
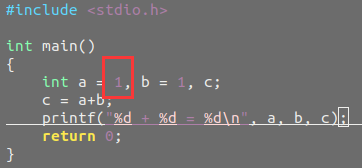
jcc foo #因为我们不关心跳转条件，这里的跳转指令用jcc代替

dec eax

foo:

call bar

其中dec eax的机器码长度为1字节，假如int3指令长度为2字节，我们在dec eax上设断点时，会覆盖掉call bar指令的第一个字节，产生一个无效指令。这样，一旦jcc foo被触发，CPU就不会到达这个断点，而是执行了foo处的这个无效指令。

如果把断点设置在指令的非首字节(中间或末尾)，会发生什么？实验表明，这个断点无效。

我是在一个A+B Problem的C程序上进行的实验，最后断点没有触发，而替换的int3指令被当作那条指令的一个操作数处理了。

猜测原因是，IA-32指令中，指令类型只占首部1-3个字节。如果int3指令不在指令类型的位置上，它就不会被解释成int3指令，而是会被当作那行指令的一个参数处理。如果是覆盖了指令类型的后面几个字节，则会改变那行指令的指令类型，但不会改变成int3。所以，如果把断点设置在指令的非首字节，则这个断点无效。

熟悉i386手册

讲义告诉我们，查阅i386手册的最高效方法是通过目录。如果目录中不能直接找到我们需要查询的名词，则通过搜索引擎了解这个名词属于哪一类概念，再在目录中查找这个概念。如果目录中有很多个章节都提及关键词，则我们通过速读小标题、图表及注解、每段的第一句话来判断哪些章节所提及的方面是我们最关心的。

必答题

1. 请在i386手册中定位以下问题的答案所在的位置。

(1) EFLAGS寄存器中的CF位是什么意思？

答：Appendix C -- Status Flag Summary

(2) ModR/M字节是什么？

答：17.2.1 ModR/M and SIB Bytes

(3) mov指令的具体格式是怎么样的？

答：Chapter 17 80386 Instruction Set -- MOV Move Data

2. （shell命令）

(1) 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？

使用命令[3][4]：

wc -l `find -name \*.h -o -name \*.c`

得到本机上的结果为100829行。

(2) 和框架代码相比，你在PA1中编写了多少行代码？

256行。

首先使用命令git log --author='17300180070'查看由我手动记录的所有本地提交，但是最前面只记载到重新组织寄存器完成之后，没有记录刚刚下载的初始状态。

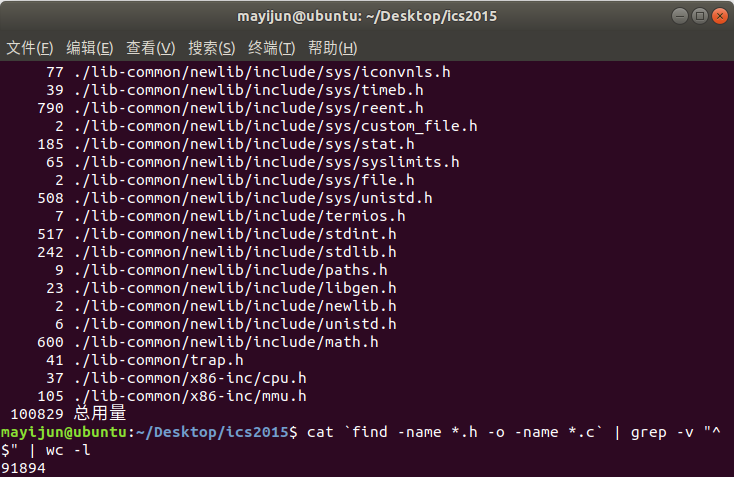
然后我使用了git log --oneline，查看各次提交信息的简略版，定位框架代码初始状态的提交ID为64d216c。

最后，使用git checkout -b PrevStage 64d216c，就成功用这一版本的代码创建了一个新分支PrevStage并切换到了该分支。然后用上一题中的办法得到代码总行数为100573，相减即得。

(3) 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？

使用命令：

cat `find -name \*.h -o -name \*.c` | grep -v “^$” | wc -l

得到本机上的结果为91894行。

3. （man的使用）请解释gcc中的-Wall和-Werror有什么作用？为什么本工程要使用这两个选项？

-Wall打开所有的有必要注意的关于程序结构的警告（当然，这些警告的内容都是容易更正的）；它也打开一些当前语言特定的警告功能。但它并不打开全部的警告功能：一般用户认为无需考虑、程序员仅仅是偶尔会检查的一些结构问题，和某些情况下不可避免的程序结构问题的警告，是不会被-Wall打开的；这些警告选项有的可以通过-Wextra打开，但大多数必须单独打开。

-Werror将任何警告都视为编译错误。

本工程使用这两个选项的目的，是强迫同学们注意代码中程序结构方面的潜在问题并更正它们（否则无法通过编译），养成良好的编程习惯。

参考文献：

[1] <http://blog.chinaunix.net/uid-21288388-id-4829098.html>

Ubuntu工作内容（十）正则表达式基础

[2] <http://www.cnblogs.com/orange1438/archive/2013/05/12/4544958.html>

scanf()正则表达式的使用

[3] <https://www.linuxidc.com/Linux/2015-02/114052.htm>

Linux 下使用 wc 统计文件夹下所有文件的代码行数（包括子目录）

[4] <https://www.linuxidc.com/Linux/2015-04/116854.htm>

Linux find 命令用法总结