PA1实验报告

赵逸凡

2019 年 9 月 23 日

1 实验进度

完成了所有内容.

2 必答题

2.1 理解基础设施

2.1.1 花费的时间

$500 * 90\% * 30 * 20 = 270000s = 75h$

2.1.2 节省的时间

$75h * (\frac{30-10}{30}) = 50h$

2.2 查阅手册

2.2.1 EFLAGS寄存器中的CF位是什么意思?

查看目录，发现2.3.4节介绍了Flags Register
2.3.4.1节(34页)指出CF(CARRY FLAG)是一个status flag，并指出附录C中有每个状态flag的定义。
在附录C(419页)得知CF“set on high-order bit carry or borrow; cleared otherwise.”

2.2.2 ModR/M字节是什么?

查看目录，发现17.2.1节(241页)介绍了ModR/M Bytes
该节指出ModR/M字节在大部分80386指令中跟在操作数后面，包含了mod field, reg field和r/m field的信息。

2.2.3 mov指令的具体格式是怎么样的?

查看目录，发现3.1节(45页)介绍了数据传送指令
该节的3.1.1小节指出mov可以进行：

- 存储器to寄存器
- 寄存器to存储器
- 寄存器to寄存器

- 立即数to寄存器
- 立即数to存储器

之间的传送，但不能在两个存储器之间传送数据。

2.2.4 selector是什么？

查看目录，发现5.1.3节(96页)介绍了这个概念

2.3 shell命令

2.3.1 查看*.h和*.c文件中的代码行数

使用命令find . -name “*.h” |xargs cat|wc -l（h替换成c以查看*.c文件）, 共有代码4165(*.c)+1311(*.h)=5476行

2.3.2 查看PA1中编写的代码行数

使用命令git checkout master回到做PA1之前的状态，此时有代码3664+1306=4970行。
因此编写了$5476 - 4970 = 506$行代码。

2.3.3 查看空行以外的代码行数

利用grep -v ^$筛选出非空行，完整命令为find . -name “*.h” |xargs cat|grep -v ^$ | wc -l
共有代码3456(*.c)+1046(*.h)=4502行

2.4 使用man

2.4.1 -Wall和-Werror有什么作用？

-Wall打开所有警告，-Werror把所有警告当作错误进行处理。

2.4.2 使用它们有什么作用？

它们可以在编译阶段把所有的fault转变为failure，虽然它们的功能很有限，但使用它们没有任何损失，不用白不用。

3 遇到的问题

3.1 找不到functioname的定义？

使用命令grep -r functionname $HOME/nemu即可。

3.2 每次make时自动调用git，自己的git commit会因为没有修改而无法提交。

使用git commit的–allow-empty选项，允许无修改提交。

3.3 为什么要用‘\\+’在模式字符串中匹配加号？

‘+’是正则表达式中的元字符，需要\来转义。而在c字符串中使用反斜杠本身是需要转义的，因此就出现了两层反斜杠。

3.4 expr.h中，格式化输入符出现了%.*s，这是什么意思？

‘*’代表这里期望一个参数。%.*s代表需要两个参数，即宽度和待打印字符串。用这种方法，我们就可以同时指定待打印字符串的长度和位置。

3.5 __attribute(aligned(PAGE_SIZE))是什么？

编译器指令，用于指定（末尾的）对齐方式。

4 选做题

4.1 计算机可以没有寄存器吗?

寄存器是CPU的一部分，读写最快。没有寄存器，就不得不使用高速缓存甚至更慢的内存。这可能导致现在取址，译码，执行，取数，写回的工作方式变得很慢或者根本不可行，因为CPU不得不每次访问高速缓存/内存来获取下一条指令的地址。

4.2 为什么全部都是函数?

使得每部分代码所做的工作更加清晰，有利于维护。

4.3 参数是从哪里来的?

在shell中执行程序时，程序文件之后的命令行参数都被传到了argv[]数组里。

4.4 reg_test()是如何测试你的实现的?

assert中使用了移位运算和掩码来得到正确值，并与寄存器中的值比较。

4.5 调用cpu_exec()的时候传入了参数-1

这个参数应该是执行的次数，输入-1会使其被强制转换成最大的32位无符号数，即执行次数的最大值。

4.6 传参-1是未定义行为吗？

有符号数赋值给无符号数会发生强制类型转换是有定义的，可预测的，写在手册里的。因此我们可以利用，不是未定义行为。

4.7 opcode_table是个什么类型的数组?

是OpcodeEntry类型的数组。OpcodeEntry是一个结构体，定义在exec.h中。

4.8 谁来指示程序的结束?

main函数结束后会隐式调用exit()函数，进行退出时的工作。我们如果显式调用exit()也可以退出程序。此外，我们还可以用atexit()函数注册希望在程序结束时被执行的函数。

4.9 NEMU是如何支持多种客户ISA的?

makefile中定义变量ISA，默认值为x86.用户指定ISA时，比对这个ISA是否在ISA列表中，如果在，则设置ISA目录为用户的选择，针对不同情况分别处理。

4.10 为什么printf()的输出要换行?

不换行会导致调试信息把正常的输出挤到一行中靠后的位置，不仅难以辨识，还会导致一些需要换行对齐的地方（如 ^ 指示符）失效。

4.11 表达式生成器如何获得C程序的打印结果?

使用了文件输入和输出,.code.C程序把输出写入一个临时文件,gen-expr.c再把它读出来，实现了两个文件之间的信息传递。

4.12 为什么要使用无符号类型?

因为所有的地址和寄存器值都是无符号数。有符号数可能有更多的未定义运算，如INT_MIN/-1.

4.13 除0的确切行为?

除0时C程序会产生一个float point exception并终止运行。

4.14 static在此处的含义是什么? 为什么要在此处使用它?

static修饰全局变量时，让这个全局变量只能在当前文件里被访问。这里使用是为了防止其他文件中出现同名的head变量，产生冲突。

4.15 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?

NEMU中这个断点好像是无效的，不会被触发。我认为是因为执行的最小单位是指令，每次判断监视点值是在一条指令结束之后。因此$pc中的地址并不是连续变化的。GDB中无法设置这样的断点，会报Warning.

5 心得与体会

5.1 关于getopt()函数

int getopt(int argc, char *const argv[]),const char *optstring);
需要使用定义在头文件unistd.h中的全局变量optarg optind opterr optopt
第三个参数‘optstring‘称选项字符串，字母后的冒号代表该参数必须带有选项；双冒号代表该参数带有可选的选项，但是参数必须与选项紧邻（无空格）；无冒号代表该参数无选项.

5.2 关于strtok()函数

char *strtok(char s[], const char *delim);
当strtok()在参数s的字符串中发现参数delim中包含的分割字符时,则会将该字符改为\ 0 字符。第一次调用时，strtok()应传递参数s字符串，之后调用则将参数s设置成NULL。每次调用成功则返回指向被分割出片段的指针。

5.3 关于数和字符串的转化

sscanf和sprintf与scanf,printf类似，它们以一个字符串而不是标准输入输出为目标。用它们可以实现整数（十进制，十六进制）与字符串之间的互相转化。另外，atoi()函数可以将字符串转为十进制整数。

5.4 封装的意义

当我们看到make_token()被expr()调用，exec_once()被cpu_exec()调用，我们就确定这部分程序给外界的接口，即需要调用的只是外层函数，而不需要调用被封装好的内层函数了。类似的,我们采取类似的思想设计函数eval()：它调用了check_parentheses()和find_main_operator()，此时我们只要考虑这两个函数的接口，不需要思考其实现细节。在编写这两个子函数时，我们才需要具体考虑其实现以适应接口。

5.5 检测匹配括号的方法

check_parentheses()函数从左往右扫过指定序列，每遇到一个左括号++cnt，每遇到一个右括号–cnt，结束时如果cnt!=0，说明表达式不合法。为了判断整个表达式是否被包在一个括号里，不能简单地由最左边是‘(’，最右边是‘)’来判断，因为它们可能并不匹配。

我的解决方法是在每次循环中检查cnt的值，一旦在中途等于0，就说明左右括号是不匹配的。

5.6 寻找主运算符的办法

find_main_operator()函数同样从左往右扫过指定序列，维护了一个in_brackets变量，记录当前位于几层括号里。按照C语法定义了每个运算符的优先级，一个运算符被选中，当前仅当它不在括号里，且是最后出现的优先级最低的运算符。

5.7 负数和解引用的处理

因为处理负数时我还没有看到监视点一节，因此使用了不一样的处理方法。

关于负数的处理：首先在eval()函数里指定遇到负号的处理办法：当遇到负号时，先用check_parentheses()判断负号以外的表达式是否合法，如果合法，再用find_main_operator()寻找主运算符，如果找不到，说明负号后面的表达式已经不需要计算，简单地return -eval(p+1,q)即可。否则仍然继续递归分解表达式。
check_parentheses()的修改：跳过负号前缀即可
find_main_operator()的修改：当且仅当一个‘-’并非紧邻在一个运算符之后时，它可以被选为主运算符。这是因为负号可能与前面的运算符隔着括号，这种情况下，‘in_brackets’保证了我们不会把一个包在括号里的负号误认为减号。其他情况下，负号一定是紧跟在另一个运算符之后的。

关于解引用：采用了讲义上的方式，在调用eval()函数前手动将tokens里的解引用识别出来。判别依据是‘*’之前出现的第一个（非括号）token是运算符。

5.8 如何解决除数为0的问题？

我让gen-expr.c生成的临时c文件.code.c写入一个日志文件，在表达式求值语句之后插入一句‘fprintf‘，写入一个标记。在gen-expr.c中，读这个日志文件。如果没有读到这个标记，说明.code.c中发生了runtime error,即除数为0。我们需要丢弃这个结果，并多进行一次循环。
这是利用了C语言中表达式除数为0时发生RE，该表达式之后的语句都不会被执行。似乎C语言不存在异常捕捉，因此使用了这个权宜之计，可能有更优美的方法存在。

当然也可以在编译选项中加入-Werror，这样所有有除0行为的表达式都会在编译阶段因为error而停止，不会被写入文件。

5.9 关于popen()函数

传递一个shell命令，该函数将创建一个管道，将命令与之关联。注意命令在这一步并没有执行，而仅仅是关联。因此，在gen-expr.c中，.code.c是在fscanf时才被执行的，不理解这一点会付出惨痛的debug代价。
这个故事告诉我们，manual才是真理，不能根据主观臆测和API猜想函数的行为。

5.10 关于监视点池

其实就是两个链表，分别维护已用结点和未用结点。只需要用链表操作在它们之间增删元素即可。