实验报告-PA2

甘晨 181240014

2019 年 11 月 18 日

此次实验未能按时完成，迟于 DDL 提交。

1 实验进程

1.1 PA2.1

反复阅读 PA 讲义，并且 RTFSC，在 PA2.1 阶段大致理解了一条指令的执行过程，以及如何完善一条指令，在完成了几个基本指令后，实现了 dummy 样例的 Hit Good Trap（然而，此时的指令并未完全实现，比如减法指令对标志寄存器的影响，还未完成）。

1.2 PA2.2

经过反复尝试，最终实现了 cputest 测试样例需要的所有指令和库函数（然而，此时有的指令的实现还存在着一些问题未被发现和解决）。

1.3 PA2.3

经过艰难地调试和 debug 过程，终于实现了 printf 的功能，再经过 RTFSC，实现了各设备功能。

2 必答题

2.1 PA2.1

一条指令的执行过程:

第一步是取指令：由 instr_fetch() 函数取指令，这里主要是取操作码（opcode），通常是取第一个字节以确定指令的操作码。但还有两种方式可以拓展操作码字节：（a）若指令以 0x0f 开头，则需再读取一个字节来确定具体指令；（b）利用 opcode 下字节的

ModR/M, 将其中的 reg/opcode 域读作 opcode 的拓展码以确定具体指令。第二步是指令译码：根据取得的 opcode 作为索引，取得 opcode_table 中的译码辅助函数（包括操作数译码辅助函数）和执行辅助函数，译码的过程实际上是根据指令取操作数，译码过程包括读取内存操作数、寄存器操作数、立即数或跳转地址的。这些获得的译码信息将被保存在结构体 decinfo 之中，decinfo 中会记录操作码（opcode），源操作数（src，src2），目的操作数（dest）以及跳转地址（jmp_pc），这些信息将提供给执行辅助函数使用。第三步是执行指令：执行过程会调用执行辅助函数，根据相应的指令读写做读写内存、寄存器以及跳转等操作，大部分指令执行过程包括对标志寄存器的更新。第四步是更新 pc 的值：原 pc 的值加上指令的长度，则 pc 会指向这一条指令的下一条指令，接下来会重复上面的过程。

实现单步执行，打印寄存器，扫描内存:

图 1: 单步执行

图 2: 打印寄存器

图 3: 扫描内存

2.2 PA1.2

词法分析:
递归求值和生成表达式检测:

图 4: 匹配规则

图 5: 识别信息

2.3 PA1.3

表达式求值功能拓展:

监视点实现:

目录定位:

Selector 的概念位于 i386 手册的 Chapter5 Memory Management 的 5.1Segment Translation 的 5.1.3Selectors

送分题:

我选择的 ISA 是 x86

理解基础设施:

基于讲义中的假设，450 次调试需要花费 9.375 天从 GDB 中获取并分析信息, 另一方面，如果市县实现了调试器，450 次调试需要花费 3.125 天从简易调试器中获取并分析信，也就是说简易调试器可以节约 2/3 的时间。

查阅手册:

CF 位首先表示 CARRY FLAG，是一种状态标志（Status Flag），这一标志受算数指令的影响，在执行算术指令之前，会改变 CF 的值；根据 Apendix C 中的描述，CF 指示高位的借位或进位，有借位或进位位 1（set），否则位 0（clear）。

ModR/M 会出现在操作码（opcode）后面，来确定获得操作数的寻址方式（specify the addressing form to be used)——内存或寄存器. 对于取内存操作数,还通过 ModR/M 来确定地址的计算方式。在 80386 指令集中，以 ModR/M 作为第二个字节的指令很常见。

根据 i386 手册，这里的 MOV 指令时 Intel 格式，不同于 AT&T 格式，其中 MOV A B 中，B 表示源操作数，A 表示目的操作数，与 AT&T 格式恰好相反。其中，源操作数可以使 8/16/32 位寄存器或内存、立即数、段基址和偏移地址做代表的内存中的内容, 目的操作数也可以来自上述内容，但不能使立即数。

Shell 指令:

共有 6304 行代码，使用的命令是:（在 nemu/目录下）”find . -name “*.c”|xargs cat |wc -l” 和” find . -name ”*.h”|xargs cat |wc -l “，最后调用之前实现的 p 指令做一下表达式求值即可。

除去空行，共有 5143 行代码，使用的命令是:（在 nemu/目录下）”find . -name “*.c”|xargs cat|grep -v $|wc -l” 和” find . -name ”*.h”|xargs cat|grep -v $|wc -l “.

使用 man:

-Wall 和-Werror 的作用是在编译程序时显示所有警告，并且把 warning 也当做 er-

图 6: 生成表达式除 0 报错

图 7: 生成的表达式

ror 显示出来，正如 PA 讲义中所说的：” 调试是从 failure 回溯 fault 的过程 “，因而尽早的观测到 failure 有可能节省调试的时间，所以说，把 warning 出来的可能出错的地方在编译时警告显示出来，有可能能避免当发现 failure 时已经很难在回溯的 fault 的情况发生。

3 实验心得

3.1 PA1.1

PA1.1 中给我留下最深的印象的就是 strtok() 函数，我在初次实现几个命令的功能时，是根据 cmd_help 指令依葫芦画瓢的，help 指令的第一步时提取第一个字符，因而我在后面的操作中学着提取了第一个字符，然而，我对传入参数 args 的理解不清，以为时包括命令标示（如 p、x、info 等等）的字符数组，因而，我谢了 char *arg = strtok(NULL,” ”) 这样一串代码，后来发现这行代码的意义不大，只是起到了分割字符数组的作用，如果指令字符后面只有一个参数，那么没必要执行这么一句，但对于 info 和 x 指令，就是必要的。但就是这一行代码，给我在 PA1.3 时带来了很多痛苦，以及消耗了我很多时间来找 bug，后面将会详述。

3.2 PA1.2

PA1.2 中的找主操作符是这一过程给我带来了体验到了一个道理：想偷懒反而可能会需要做的更多，一开始 PA 讲义上说可以把整个 tokens 数组遍历一遍来找，一开始，我想过可以给每个操作符一个优先级，方便比较，可想来还要把 tokens 数组遍历一遍，并且还要新定义一个结构体，有点麻烦，所以想另寻他法。最终，我想到了一个” 好办法 “，我用递归的方式，先从 tokens 右边往左找，找到不在括号内的第一个运算符，如果是 + 或-，那么就返回这个运算符的 index，如果是 * 或/，那么再从左往右找不在括号里的第一个运算符；如果是 + 或-，那么返回包含左边第一个运算符但不包括右边第一个运算符的数组再去递归查找；如果是 * 或/，那么先记下右边那个运算符的 index，在递归查找包括右边运算符但不包括左边运算符的数组，若返回的 index 变了，那么就去改变后的 index，若不变，那就取那个 index；查找最后都会有 i == j, 若 i>j, 则 assert(0). 在 PA1.2 中，这一方法成功了，而且也通过了生成表达式的样例测试，也就是这成功的方法，使我在面对 PA1.3 时感觉非常糟糕。

图 8: 表达式求值结果

图 9: 表达式求值功能拓展

3.3 PA1.3

好吧，似乎 PA1.1、PA1.2 都给 PA1.3 挖了坑，对于 PA1.2，我发现我找主操作符的方法在 PA1.3 中根本没法拓展，加了几个运算符之后，这个方法就很难行的通了，没办法了。在还是那位大佬（就是教我调整寄存器结构的那位大佬）的指点下，我还是利用了拷贝 tokens，并在结构体内加优先级的方法来做，出乎意料的是，这个方法竟然异常简单，我本来将近百行的代码瞬间缩短为了小几十行，我本来想偷懒的想法反而让我花费了更多时间。到 PA1.1 了，唉，没脸说了，2 我花了一个完晚上和周六一整个白天都在想这个 bug 是怎么回事。当时情况是这样的，我在拓展表达式求值的功能，因此，需要匹配寄存器，也就是需要匹配 $ 字符，我先是根据我的想法，写了正则表达式，我用 $edx 去测试，可报错为” position 0 no match “，而且这个 position 0 指向的是 “e”，当时我并没有注意到这个问题，又试了几种，还是不行，后来，我又上网查询，向大佬（还是那位大佬）求教，然而按照大佬的匹配方式，居然还是报错了！我都惊呆了！以为是虚拟机与真机有差别而产生了 bug!（似乎有点好笑），没办法，我后来只好先不实现寄存器，接着往下走，于是我开始搞其他的。偶然间，我打开了 ui.c 文件，就是写指令的文件，还记得我之前说的对 strtok 理解不清吗，我在取 p 指令的表达式的时候，也多次一举的又 strtok 了一下，我想我需要把后面的表达式与前面的操作符表示分开，而且后面的表达式需要时一整块（我当时还不知道传进来的字符数组已经去掉指令标志了），所以我不能按空格分割，而需要按一个不可能出现的字符来分割，你猜我选择了那个字符（QAQ），教训相当惨痛!!!

4 鸣谢

由衷感谢那位 x 姓大佬在我在 PA1 的泥潭中奄奄一息之时给予的帮助和指点，在此表示深深感谢。

图 10: 监视点结构

图 11: 监视点管理