PA#0, 1, 2, 3

1511186
梁宸

February 25, 2020

github: https://github.com/bsblcc/nemu

环境: nemu 2018版本

pa1, 2: deepin linux x64, gcc 8.3

pa3: lubuntu x64 in Virtualbox, gcc 7

1 PA#0

1.1 必答题

学习makefile, gdb等其他工具.

2 PA#1

2.1 实现正确的寄存器结构体

查看CPU_state结构体的原始定义, 以及/nemu/src/cpu/reg.c中的reg_test函数中的assert, 可知寄存器结构体的实现需要能以两种方式访问模拟的寄存器文件: gpr数组或者eax等单独的变量, 并且两种方式要对应相同的内存, 因此这里需要使用一层union; 同时, 通过gpr数组能访问寄存器的不同部分, 它们也应指向同一内存, 因此gpr的定义中也应有一层union.

2.2 究竟要执行多久?

当传入参数为-1时, cpu_exec中的主循环不会被执行. 这意味着nemu只会进行cpu的初始化工作, 然后进入停止状态, 并不执行任何指令.

2.3 实现单步执行, 打印寄存器, 扫描内存

查看代码, 可以发现NEMU接受的command在ui.c中的cmd_table数组中, 于是在这里注册要新增的调试命令的名称、描述和handler. 主要任务是实现几个handler函数来完成具体的调试功能.

2.3.1 单步执行

在上面的问题中, 发现cpu_exec函数的参数n可以决定cpu主循环执行次数, 即单步执行的次数. 因此只要在单步调试的hanlder中以此调用cpu_exec函数即可.

2.3.2 打印寄存器

同样在上面的问题中, 发现可以用cpu.eax等方式访问寄存器文件, 因此在打印寄存器的handler中将它们打印出即可.

2.3.3 扫描内存

查看NEMU的代码结构, 内存有关的头文件在/include/memory/memory.h中. 查看它提供的内存访问接口, 我们需要的应是vaddr_read函数, 再查看其实现, 发现参数addr对应要读的虚拟地址[1], 参数len对应要读的字节数. 该函数会进行一些参数合理性检查, 最后从pmem数组中访问NEMU的内存. 因此在扫描内存中的handler中, 先计算得到地址表达式的值, 再以地址和长度调用vaddr_read函数即可实现内存打印. 在实现进行到这里时, 尚未要求实现表达式值计算的函数, 因此目前暂且先实现了一个stub, 它对于任何表达式都返回值0x100000.

2.4 算术表达式

2.4.1 实现词法分析

查看代码可以看到NEMU的词法分析十分粗暴, 它将词法的正则文法用C的regex库函数表达, 每次穷举尝试匹配所有规则. 这种方法实现起来相对简单, 值得注意的一点是, 在正则表达式的约定中, '
'字符用来转义, 而在C语言中它又需要转义, 因此嵌套需要'

'; 另一点是在成功匹配到某类词后, 还需要把特定的信息存起来, 比如整数和寄存器等, 并且事实上在这一步中, 我们已经可以分辨出减号和负号、乘号和解引用符, 因为只需要检查它们的前一个符号是否为右括号、数或寄存器.

2.4.2 实现递归求值

文档中介绍一种基于运算符优先级和递归的语法分析方法来处理特定的表达式求值, 并且算法思路已给出, 实现不难.

具体实现中, 先检查递归的终止条件, 即是空串或仅有一个单词, 并且这单词应有值, 比如寄存器、常数等.

[1] 这只是逻辑上的直接映射, 还没有涉及地址翻译.

接下来检查括号的合法性, 并且去掉最外层的括号. 简单的合法性检查用一个栈即可, 合法性等价于最后栈为空. 有趣的一点是考虑一次消除所有最外层的括号, 形如'(((((())()(()))))'这种情况. 这时可以一遍扫描中记录每对匹配括号的区间, 那么一个括号是最外层的, 当且仅当它的区间包含所有其他括号的区间, 这等价与当前左端点最小的区间等于右端点最大的区间. 因此在一遍扫描后, 分别按左端点和右端点对所有区间排序, 再扫描一次[2].

最后根据主运算符进行分裂. 主运算符应当是在括号外部, 并且优先级最低的运算符, 因此扫描过程与检查括号时类似. 其中是单目运算符在词法分析时就已经区分出了, 它们的优先级是最高的, 因此如果它们是主运算符, 且表达式是合法的, 那么就可以确定左侧没有子表达式, 因此只需分裂到右边.

2.5 监视点

2.5.1 实现监视点池的管理

此处主要的工作是实现两个链表分别管理空闲和占用的watchpoint, 具体而言实现了带dummy node的双向链表. 则free和new基本上是对称的操作, 在两个链表中插入删除即可. 并且watchpoint的链表几乎只是逻辑上的结构, 没有像malloc那样关联内存块, 因此插入的顺序只会影响下标, 没有效率问题.

2.5.2 温故而知新

在C中static修饰全局变量和函数的目的一般是保护内部实现, 禁止来自其他文件的引用.

2.5.3 实现监视点

文档中的说明已经十分详细, 值得一提的是当多个watchpoint被触发时, 我的实现是报告所有.

2.5.4 一点也不能长?

事实上文章中已经给出答案了, 不能. 因为若int3指令大小超过一个字节, 那么当覆写到目标地址a时, 可能会覆写多条短的指令, 而中间的那些可能是某些跳转的目的地, 因此有可能会跳转到覆写的int3指令中间, 执行无意义的代码.

2.5.5 "随心所欲"的断点

仅凭猜想, 如果int3覆写到了某条指令的中间部分, 那么该条指令的结构会被破坏, 执行到此处时会执行无意义的指令, 无法预料后果.

[2]虽然对于非空值表达式这个情景并不多见, 但还是一个不错的娱乐.

2.5.6 NEMU的前世今生

模拟器调试是尝试模拟系统级的行为, 调试时可以从更高层[3]的系统级获取信息. 而调试器是在用户级, 需要向系统请求用于调试的系统调用等资源, 比如ptrace, 调试器需要不停的进行IPC, 调试器和被调试程序是基本平等的, 最多就是子进程和父进程. 从用途上来讲, 调试器一般仅用于调试host的程序, 而模拟器(系统级)的目的是模拟一个完整的计算机, 包括IO等, 而且经常会包括target到host的二进制代码翻译.

2.6 i386手册

2.6.1 尝试通过目录定位关注的问题

selector多指段选择子, 是与内存分段有关的概念, 因此可以在内存管理章节中的分段机制部分找到它. 或者因为它不算是常见的单词, 也可以全局搜索selector找到. 目录中指明关于selector的内容集中在96页.

2.6.2 必答题

查阅i386手册

EFLAGS寄存器中的CF位是什么意思? EFLAGS寄存器是标志寄存器的名字, 因此在目录中搜索不到EFLAGS的情况下, 搜索flags register. 找到对应章节在P33. 这里介绍CF的含义是Carry Flag, 但没有详细的介绍. 再全局搜索CF, 在p50有关算术运算中介绍它是进位标志, 当加减法最高位需要进位设为1.

ModR/M字节是什么? 仍然直接搜索ModR/M, 发现它在指令结构有关的章节中, 位置是P241. 但印象里它是i386指令编码的一个补足, 好像是index寻址等较复杂的指令会用到的, 涉及到整个指令模式的设计, 因此应该要阅读17.1和17.2或者更多.

mov指令的具体格式是怎么样的? i386手册的后面给出了所有指令的格式, 找到mov在p345.

统计代码行数 上网寻找资料后学到了grep工具来统计代码行数, 它是全局文本正则匹配的工具, 很强大. 空行即是^$. 在第一次编译后的commit处创建一个新的分支, checkout到那里, 再统计此版本的代码行数, 就得到了框架代码行数, 做个减法得到我编写的代码行数. 目前这个数字是588. 框架代码行数是不会变的, 因此只需统计一次, 硬编码即可. 将上述过程写成sh文件, 加入makefile豪华午餐.

编译选项 -Wall的含义是warn all, 即警告所有gcc能发出的警告; -Werror的含义是将warning视为error. 这两个选项无疑提高代码效率(主要是编译时的效率), 并且减少一些隐蔽bug的可能性, 比如类型宽度等. 开启它们会使人写出严谨的代码, 尤其是在一些大型系统中. 虽然我一开始就把werror注释掉了.

[3]或者说更底层.

3 PA#2

3.1 RTFSC(2)

3.1.1 立即数背后的故事

对于模拟器来说, 这两种情况都是host和guest的大小端不同. 此时我们要分别弄清host和guest的体系结构. 对host来说, 可以用gcc的一些宏, 在编译的时候得知host的大小端, 这样可以在内存相关的部分使用条件编译. 或者用一些简单的测试函数, 在NEMU运行时判断; 对guest来说, 通常会提前获知guest的体系结构. 像文中所举的例子里, 我们只针对Motorola 68k作为guest. 我们也可以在可执行文件中获得target的体系结构信息, 无论如何, 我们一定会知道目标程序对应哪个体系结构, 这样就又可以使用之前的两种办法了. 一般而言, 像qemu等项目都会使用条件编译的方法.

当host和guest的大小端相同时, 我们不需要做额外的工作; 但当二者不同时, 当从guest code中读多字节的内存时, 需要以相反的顺序读, 虽然这听起来相当慢.

3.2 运行第一个客户程序

3.2.1 整体框架

这部分的主要难点在于NEMU代码的实现结构, 和80386的指令编码, 而二者关系紧密, 所以花费了较长时间弄清这两部分.

80386编码的大概思路是, 通过前面二或多个字节的opcode(或者说main opcode)来确定指令的基本形式, 比如是否有前缀(有的话就要接着取一个字节), 是否有modR/M字节等. 其后可能有立即数位等, 但除了opcode, 其他都不是必要的. [4]

modR/M字节是指令的拓展. 其中中间的reg/opcode域, 只能有两种解释之一: reg即表示一个寄存器编号, opcode即extended opcode, 它作为opcode的补充, 和main opcode一起构成opcode. 文档中解释第二种情况是由于指令种类过多, 于是将类似的指令集中为group, 它们main opcode相同, 由ext opcode区分; 头部的mod域和尾部的r/m域中, mod表明r/m的意义: 内存或者寄存器, 如果是寄存器(mod为11), 后面的r/m就被解释为一个寄存器编号. 有些情况下, mod也会和后面的SIB字节一起做scaled indexed的复杂寻址.

NEMU的具体实现使用了比较结构化的设计. 译码和执行阶段使用了DHelpers (Decode)和Ehelpers (Execute), 其中DHelper又被进一步分解为各个DOHelper (Decode Operand). 对应到上文的指令编码, DHelper主要对应的是main opcode, 它比较模糊, 比如它不会区别ModR/M实际内容是什么, 都被视为operand; 这些具体的识别和取出operand的工作由DOHelper进行;

[4] 这种CISC指令结构复杂, 长度不定, 甚至NEMU仅作为模拟器的代码中, 都能看出它的各种弊端, 对各阶段层次的破坏, 复杂的控制逻辑等等. 很难想象Intel的电路设计会有多恐怖, 也难怪除了x86, 大家都投奔RISC了.

而对operand的抽象可以使不同的指令执行阶段的逻辑也复用, 也就是许多EHelper. EHelper尽量要用RTL去实现. 因为NEMU是模拟器, 我们有很多更灵活的C语句去代替RTL指令[5], 但是这不利于中间层的构建, 毕竟RTL就是gcc的IR. 而且RTL也可以类比x86里面的微操作, 显得更真实一点.

另外对于某些无法一次判断opcode的指令, NEMU的做法是在EHelper中再次调用别的EHelper, 比如之后要做的sub指令, 就是先调用group的DHelper, 得到operand和ext opcode, 再根据它调用sub自己的EHelper.

3.2.2 添加指令

了解了上述内容, 如果要添加新的指令, 就先要映射对应的DHelper和EHelper. 其中DHelper以及下属的DOHelper都已经给出实现了. 所以要做的就是在80386的手册上, 找到指令的main opcode, 然后根据17.2节的具体格式, 和附录A的图, 辨别它该用哪种DHelper. 然后实现EHelper, 要参考80386手册部分的伪代码, 并且最好先实现各种RTL伪指令, 用它们. 在all-instr.h下声明EHelper. 为了执行到dummy的good trap处, 需要添加的指令在文档上已给出, 在build目录下也有事先反汇编好的代码, 对debug帮助很大.

要注意的是eflags上标志位的更新, 以及RTL程序中寄存器的保护, 这有点类似汇编编程, 但没有汇编那么明确的调用约定. 此外NEMU的CPU主循环与真实的CPU不同的是, IP寄存器不在取指后更新, 而是在执行阶段完成后再更新, 因此在call指令中push的应该是seq_eip, 即下一条指令的地址.

比较疑惑的就是80386手册附录A中指明指令CALL(E8, near)的addressing mode是A, 即绝对地址, 但是这违反17.2节的内容和常理, 应该是J更合理, 事实上我的实现中也是这样work的.

3.2.3 RTL寄存器中值的生存期

从短到长, t0的生存期是一个RTL伪指令内部; t0-t3是一个执行阶段, 即一个EHelper内部; id_src, id_src2和id_dest是一个取指和执行阶段; 各个通用寄存器的生存期几乎是整个程序.

3.3 程序, 运行时环境与AM

3.3.1 为什么要有AM?

就目前的AM而言, AM与OS提供的运行时环境的区别在于, AM的要更简单更直接, 比如AM提供的是单任务环境等等. 但就整体的系统框架而言, 我觉得AM能做的事情和OS差别不大.

[5] 比如直接把Rtlreg类型当做int计算.

3.3.2 堆和栈在哪里?

如果把堆和栈的位置交给程序自己决定的话, 对于目前的AM来说, 风险比较大, 很可能无法管理, 比如程序set到了mmio的部分. 因此AM在链接脚本里设定了堆和栈的位置.

3.3.3 实现更多的指令

这部分的工作和之前一样, 只是工程量较大, 而且debug比较麻烦. 所以我先提前完成了下一节的qemu diff-test, 再加上gdb以及之前nemu实现的debugger一起使用. 值得一提的是, 之前rtl的调用约定不完善的问题在这里暴露出了. 比如rtl_setcc, 它不是rtl伪指令, 但也会被其他rtl指令调用, 同时并没有明确的约定指明这时rtl寄存器该怎样使用, 比如谁该保护哪些寄存器; 同时rtl伪指令中只能使用at寄存器, 所以寄存器数量有时很紧张, 容易出现memory aliasing的bug, 我在这里又加了一个at2寄存器, 但它多半可以被优化掉.

3.3.4 指令名对照

当ATT和INTEL的汇编指令名不同时, 直接参照opcode, 因为它是唯一的, 这是由处理器决定的.

3.3.5 实现字符串处理函数

就是参照lib manual的行为实现这些函数.

3.3.6 实现sprintf

查看代码和手册, 得知sprintf应该调用vsprintf实现输出, 自己只做va_list的分析. 这里只实现了其他%s,%d和其他非转义字符. %s可以通过strcpy实现; %d我这里简单实现了一个丑陋的dec到字符串的转换函数, 写得我都不忍看了.

3.4 基础设施(2)

3.4.1 如何生成native的可执行文件

以app来说, 先执行app目录下的makefile, 它include了makefile.app, 可以看到有三个组成部分: am, app和lib. 其中am是在其目录下的build脚本编译的, native的arch下使用的是g++. 而app和lib都最终靠makefile.compile编译, 视类型使用gcc, g++和ar. lib文件通过ar打包成.

3.4.2 奇怪的错误码

makefile的错误码和shell执行的exit code相同. test失败的话是assert失败, 这时shell指令的exit code应该是1.

3.4.3 实现DiffTest

qemu的reg类型在protocol.h下有定义, 因此按照它, 将qemu和nemu的各个寄存器内容比较即可. 从软件工程角度讲, 这个test方法也许可以说是整个pa2的核心环节了, 会节省相当多的时间.

3.4.4 捕捉死循环

如果要识别死循环, 首先要识别出循环, 可以先把guest code划分成基本块, 参照qemu的jit实现, 在动态翻译的情况下, 基本块(bb)只被跳转指令划分, 那么在这种bb构成的控制流图(cfg)中, 我们的执行路径一旦出现了环, 等价于访问到了已被访问的bb, 就出现了循环;

而死循环和正常循环的区别在于, 决定是否跳转的值, 是否是个循环不变量. 那么就要基于数据流图(dfg)再分析, 如果这个值在dfg上路径的所有产生数据依赖的值[6], 在相邻两次循环中都不变, 那么它就是不变量, 因此这是一个死循环. 由于开销等问题, 可以设定在循环了某个次数之后才开始死循环判断. 但这种基于语法的方法应该只能解决较少的死循环, 没法处理由于语义造成的, 比如while(i++)这种, 这等价于停机问题, 目前是被证明为不存在, 虽然这个例子中很可能因为i溢出而停止.

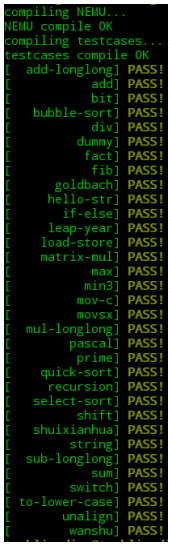

Figure 1: PA2第一部分结果

[6]可以做一次图的遍历得到.

3.5 输入输出

3.5.1 理解volatile关键字

volatile关键字, 如其字面意思, 表示一个变量是易变的, 对c编译器来说, 意思就是它可能会在编译器无法预料的时刻改变值. 一般来讲编译器能预料的, 就是在它编译的这段程序中的写操作. 而此外, 比如像这里nemu的设备寄存器, 会在不确定的时刻被设备写. 一般来讲就是有并发控制流的情景会产生这个需求.

对编译器来说, 这个性质很重要, 因为一旦某个变量是volatile的, 那么如果把它cache到寄存器里, 就完全无法保证coherence. 因此编译器不能对它进行寄存器分配等类似的优化, 只能按照原本的语义, 从那个位置读写.

3.5.2 运行Hello World

port IO的整体代码框架大概就是: nemu用线性表的形式维护设备注册的端口, 每个端口对应一个回调函数, 这里是相当于做了串行的简化, 毕竟是在模拟, 只当cpu真正去访问那些端口时, 模拟出来的设备才被回调, 懒惰地响应. 用户程序, 或者说最终是库, 调用in/out指令访问某个端口. 这种指令在nemu中会被翻译为pio_read和pio_write系列函数, 它们在读/写相应位置端口的先/后, 会调用回调函数, 来提醒设备发送/接受. 在此基础上, 实现不难.

3.5.3 实现printf

比较好的方法可以像之前定义Helper那样, 用宏把printf和vprintf相似的逻辑合起来, 而区分他们的输出方法(字符数组和_putc). 但我这里为了偷懒, 就直接用vsprintf输出到一个字符数组里, 再遍历去_putc.

3.5.4 实现IOE

可以参考native下的timer, 和nemu中模拟的timer硬件的逻辑, 得知它的意图是要把lo设位in指令获得的32位整数, hi位是0.

3.5.5 看看NEMU跑多快

这部分是跑几个benchmark, 一开始我配置没变去跑, 慢得甚至跑不完. 关闭了diff test和debug后, 性能依然很差. 又想到之前为了gdb调试, 把所有的编译选项都改成了-O0, 改回-O2后, 又跑了100分左右, 还是很差. 不过这倒给之后留了很多优化空间. 而且尴尬的是我native下的cpu和文档中的差不多, 但跑分只有一半不到, 虽然我开了gdb.

3.5.6 如何检测多个键同时被按下?

有了通码和断码, 那么无论cpu以哪种方式–外部中断或者是像现在这样主动轮询–得到键盘信号之后, 对应通和断按位操作, 就能得到当前正在被按下的键的集合S. 像格斗游戏那些判定按键组合的顺序, 就可以归纳为一个自动机, 每次S的改变都是一个输入. 不过我查了一下资料, 似乎他们游戏行业内的人都是用二位数组把出招表存起来, 当连续按键数超过某个阈值后开始遍历...

3.5.7 实现IOE(2)

keyboard的实现跟上面差不多, 甚至可以照搬native下的实现, 多亏良好的代码框架, 接口很统一.

3.5.8 实现IOE(3)

屏幕的大小信息在代码中可以看到, 是高16位和低16位分别存储长宽. 然后实现就不难了.

3.5.9 添加内存映射I/O

就是在物理内存访问中多判断一下目标地址是不是mmio的区域, 文档中已经说得很详细了.

3.5.10 实现IOE(4)

查看代码, 数组fb对应的就是vga那块内存. 这样当user想要绘制点什么的时候, 用klib时, 最终就要把_FBCtlReg对应的显示信息输出到fb上. 还是可以参考native下的实现, 由于是像素矩阵是行主的, 因此每次要memcpy一行像素, 同时因为显示屏(或者说vga内存)是有边界的, 而user提供的参数可能会出界, 所以还要判断一下取个最小值. 另外sync参数的作用, 应该是什么也不做, 让显示器保持原来的像素? 最终nemu那里是用了SDL库来监视这段vga内存, 值得注意的是在nemu处它也是0x40000, 这说明nemu和运行的guest code要有这样严格的约定, 否则无法做vga输出.

3.5.11 展示你的计算机系统

文档中说做到这里就可以运行两个小程序了, 不过刷新像素的速度很慢.

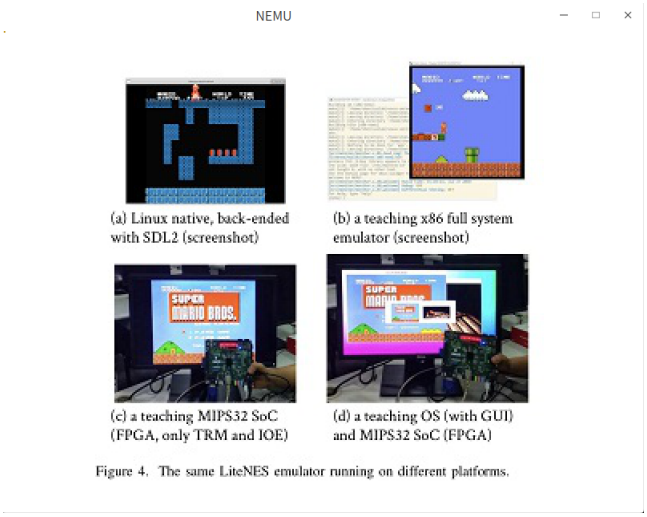

Figure 2: PA2第二部分结果

3.5.12 必答题

编译与链接 只有static和inline都去掉的情况下会链接失败. 在nemu的编译结构中, rtl.h会被多个.c文件包含, 而这些c文件又会编译成不同的.o文件, 最后链接到一起, 这时会链接器会发现多个rtl.h中的函数, 因此问题的关键就在于, 编译器是否认为它们是多重定义.

在c中, static修饰的函数会被认为是局部的, 而inline的情况比较复杂, 和语言标准有关, 但由于rtl.h中的那些函数, 并没有在文件外部被引用, 所以即使只有inline关键词, 也是能链接成功的. 但若是两个关键词都没有的情况下, 这些函数就会产生冲突, 链接器无法确定该链接哪个. 可以仿照nemu的结构写一个简单程序验证:

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l     F .text  0000000000000007 fun_static_inline
0000000000000007 l     F .text  0000000000000007 fun_static
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
000000000000000e g     F .text  0000000000000007 fun_extern_inline
0000000000000015 g     F .text  0000000000000007 fun_extern
000000000000001c g     F .text  0000000000000029 main
0000000000000000         *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*  0000000000000000 fun_inline
```

Figure 3: test样例

这个结果是gcc和clang得到的, 用的是gnu的标准. 可以看到, static修饰的函数被认为是局部的, 因此在链接时, 不会产生多重定义. 而inline单独修饰的函数比较特别, 编译器不会将它归为任何符号, 而只是一个代码段.

编译与链接.

1 在common.h中添加一个volatile static变量后, 会有29个实体. 这个结果可以通过objdump出nemu的可执行文件的符号表看到. 原因和上一问中相似.

2 在debug.h中添加相同的声明后, 结果仍然是29不变. 这是因为debug.h中包含了common.h, 所以不会有只引用debug.h而没有引用common.h的.c文件, 因此实体数量不会增多.

3 如果对变量进行初始化, 就导致它从一个弱符号变为强符号, 这时在任意一个同时包含了两个头文件的.c文件的编译过程中, 都会产生重定义.

了解Makefile. 这个过程和上个问题说的差不多. nemu的make过程是先编译.o中间文件, 这是通过搜索所有src目录下的.c文件得到的, 而build时的依赖由obj目录下的.d文件指明. 然后再把这些.o文件链接成最终的nemu文件.

4 PA#3

4.1 穿越时空的旅程

4.1.1 特殊的原因?

我认为触发中断时的上下文不能由软件来保存. 如果是由软件完成, 那么它应该也需要跳转到某个handler去保存上下文, 这和触发中断时相似, 都要面对同样的上下文保存过程, 所以最终应该还是要由硬件来保存; 另外如果由软件来保存上下文, 会有很严重的安全隐患, 比如恶意程序修改push eip, 让handler返回时跳转到其他的恶意程序.

4.1.2 实现i386中断机制

实现lidt, 它将idt的base和limit从内存中加载到idtr中, 按照i386文档实现即可, 要注意的是在16位情况下, 需要加载24 bit的base, 注意长度.

实现int, 这个直接调用raise_intr就行.

实现raise_intr, 它是cpu执行到int之后的处理流程, 如文档所描述的, 先把一些上下文push了, 然后访问idtr寄存器获得idt的地址, 再根据中断序号得到idt中的一个描述符. 描述符中包含handler的地址, 也就是在trap.S中定义的那些. 需要注意的是idt的描述符中, 跳转的目标地址是拆成两部分分别存在头尾的, 所以实现时要读两次再合起来.

4.1.3 重新组织_Context结构体

首先实现pusha指令, 按照i386手册里实现即可.

对于_Context结构体, 我们的目标是在irq_handle函数开始时, 栈上的内容顺序和_Context结构体的成员定义顺序相同[7]. 而这时栈上的内容由raise_intr过程, 和trap.S中的idt handler过程决定. 查看它们中push和pusha等指令的使用, 就能知道其顺序.

4.1.4 实现正确的事件分发

较简单, 按照文档实现即可, 就加几行判断. 从trap.S跳转到irq_handler后, 识别并分发给os.

4.1.5 恢复上下文

较简单, 按照i386手册实现popa和iret指令即可.

4.2 用户程序和系统调用

4.2.1 实现loader

此时的loader的功能是将ramdisk的内容整个加载到指定位置, 因此调用ramdisk.c中提供的两个函数, get_ramdisk_size获取ramdisk的大小, ramdisk_read读即可.

4.2.2 系统调用的必要性

不能把AM中的API直接放给用户程序, 这样会使user权限过大, 比如任何一个用户程序都能改变系统时钟, 改写键盘状态, 永远显示gui而无法被最小化等.

4.2.3 识别系统调用

开始实现syscall. 从user开始, user调用syscall是通过libos中的nanos.c中的API, 它会把参数按照约定保存到几个寄存器中, 然后执行int指令. 然后nemu执行到这个int指令, 会走一遍上述的中断流程, 最后到CTE中的irq_handler中分发, 在这里syscall应当被识别出, 然后在OS的do_event中调用do_syscall进行处理, 最后在do_syscall中再分发. 按照上述流程检查每一步的实现即可, 注意user和hardware/OS流程的对偶, 参数要一致.

4.2.4 实现SYS_yield系统调用

实现GPR宏, 按照syscall中的寄存器和参数顺序的约定, 为eax, ebx, ecx, edx.

添加SYS_yield调用. 在do_syscall中的分发中识别出它, 然后按照文档所述调用_yield.

[7]因为C中struct member定义的顺序和内存分配的顺序一致.

设置系统调用的返回值. 它要存在eax中, 在arch.h中约定用GPRx表示, 因此存到这里就行.

4.2.5 实现SYS_exit系统调用

如文档所述, SYS_exit会用它收到的唯一一个参数调用_halt来停止.

4.2.6 在Nanos-lite上运行Hello world

实现SYS_write, 流程和上述其他syscall一样. 最终的区别在于syscall handler, 对于目前的write, 文档的要求是输出到stdout, 因此调用klib中的_putc即可.

4.2.7 实现堆区管理

实现系统调用SYS_brk. 对于SYS_brk, 目前这是个批处理系统, 所以其实它总会返回0, 但应当还要做一些合理性检查, 比如nemu实质上vaddr和paddr是线性映射, 所以brk的参数不应该大于那个上限.

实现用户端的_sbrk, 它的具体逻辑文档中已经说的很明白了. 大体是让brk上升一个增量, 然后尝试SYS_brk新的值, 如果成功, 就返回旧brk值; 否则返回-1. 值得注意的一点是brk初始值由_end符号的地址来指示, 它在链接时被确定. 而我没有找到一个好的办法来赋brk变量的初始值, 即_end, 它不被gcc认为是个常量. 所以我在每次执行_sbrk时都检查一次它是否需要赋初始值.

4.3 文件系统

4.3.1 让loader使用文件

主要是实现fs.c中那几个函数. 对于fs_open, 主要功能是查找文件表, 返回参数文件名对应的序号作为fd; 对于fs_read, 此时所有的file都假设在ramdisk上, 因此调用ramdisk的API去读. 同时注意维护file的offset, 要从offset处读, 实际读了多少offset就增加多少; 对于fs_close, 文档中说可以永远返回0, 因为正常的OS是要维护一个open file table的, 目前我们没有, 所以随便close. 值得注意的是, 要在read时判断参数len是否会超过file边界, 如果超过了就要修改实际读的长度.

实现了这些之后, 在loader处就可以调用fs_read了.

4.3.2 实现完整的文件系统

实现fs_write, 它和上述的fs_read几乎相同; 实现fs_lseek, 查man得知它的参数指明3个操作, 而且它们似乎和字面意思差很多... CUR操作把当前的offset加上某数; SET操作把当前的offset设置为某数; END操作把当前的offset加上文件大小.

4.3.3 把串口抽象成文件

开始实现虚拟文件系统, 首先在文件表最前面加上串口对应的项, 其中最重要的是仿照OOP的read和write方法. 对于串口写来说, 它的write方法就和之前一样, 由_putc完成. 随后fs_write和read中就要判断这个file是不是真的在ramdisk, 即有没有显式的write和read方法, 如果有, 它就是一个虚拟文件, 要调用它的方法去读写.

4.3.4 把VGA显存抽象成文件

首先要在init_fs处, 设置显存文件的大小, 这个可以通过AM中的screen_height和width来获得.

随后, 用户程序首先应当先要获取monitor的大小, 它要通过访问dispinfo文件来获得. 当用户fs_read到它, 就会调用它的read方法dispinfo_read, OS会从一个字符串dispinfo中read内容. dispinfo是在init_device时提前准备的, 格式由文档中给出, 具体长宽参数也是由AM的API获得.

用户程序获得了大小, 就可以开始绘图了. 这个过程被抽象为向显存文件中写, 它会调用方法fb_write, 最终由AM的API draw_rect完成. 要注意的是, fb_write的参数意图是线性地写, draw_rect是按一个矩形绘制的, 而AM内部的实现(PA2)还是按照线性写的, 而用户调用fb_write时可能绘制的不是一个矩形, 因此这里还要切割一下, 而且中间这个矩形API就显得很多余. 考虑到AM内部的线性实现, 我这里的切割方法如下图, 最多有三块.

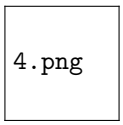

Figure 4: 切割方法

还要注意一点就是API中x, y和长宽的对应, 我一开始写反了, 输出图像是个镜像, 但那个logo我没看出来... 而且这里些奇怪的问题, 在NDL, 也就是这里的用户程序, 的代码中, 有一行是打开event文件, 而这应该是下面要实现的内容, 不知道是什么情况; 另一个也是在用户程序中, libc里的fgets函数似乎没法识别出文件的终止, 我修改了很多次dispinfo_read中的返回值定义, 还是没有找到问题. 所以我最后改了这部分的代码, 如果fgets中读的指针超过了文件大小, 就认为读到了文件结尾, 停止.

4.3.5 把设备输入抽象成文件

主要意图是, 当用户读event file的时候, 如果目前有键盘信号, 就返回对应的事件; 如果没有, 就返回当前启动后经过的时间. 因此添加这个虚拟文件, 实现它的read方法. 而键盘和时钟对应的API和一些约定的宏, 在PA2的内容中都实现过, 查看那些代码就能找到定义.