PA4 实验报告

匡亚明学院 赵超懿 191870271

注：因为重新提交了之前的报告，故把前面的报告也整合到PA4报告的后面。

实验进度：

我已完成PA4 全部内容 （PA4 OJ 报错 Non UTF-8 output，但是本地都可以运行）

必做题：（架构x86）

分时多任务的具体过程 请结合代码, 解释分页机制和硬件中断是如何支撑仙剑奇侠传和hello程序在我们的计算机系统(Nanos-lite, AM, NEMU)中分时运行的.　　答：首先在nanos中将pal的二进制文件加载。在加载过程中操作系统通过map函数建立好虚拟地址到物理地址的映射，在加载好用户进程后，通过yield切换进程控制块。然后通过调度函数切换到用户进程进行执行。在这个过程中nemu执行指令，am中通过cte切换到用户进程。

其中nemu分页运行机制：在nanos中建立映射后，设置了cr3寄存器和cr0寄存器的PG位，执行用户程序时，每当访问一个虚拟地址，cpu会先进行page table walk，取线性地址的高10位，得到页目录，然后取中间十位得到页表，加上offset得到实际的物理地址，进行访问。这样用户进程就可以运行在虚拟地址空间上了。

pal和hello同时运行：（假设当前运行的是pal）首先在每次执行指令后，nemu检查是否有时钟中断到来，如果有时钟中断到来且处于开中断的状态，调用raise_intr，然后检查特权级，发现是当前处于用户态，于是先通过TSS机制取得内核栈的栈帧，保存用户栈的栈指针，然后将内核栈的栈指针赋值给esp，然后将ss段寄存器和用户栈栈指针push到内核栈，然后依次push eflags，cs，pc，通用寄存器等，进入cte，在cte中先获取当前虚拟地址空间寄存器cr3，然后处理yield自陷事件，此时使用的都是内核栈，切换到hello的进程控制块（current指向），然后开始恢复hello的上下文，pop出通用寄存器，然后执行iret指令，弹出pc，ss，eflags后检车cs的RPL，发现应当返回用户栈，然后接着pop出用户栈的esp和ss，然后在用户栈上执行hello，至此就完成了栈的切换。

实验过程以及实验心得（包括一些我理解的蓝框题的回答）：

这次是做一点写一点报告（=v=）

PA 4.1

第一个框并不难，看了讲义的图解都清楚了，再看看native的代码，啪的一下，很快啊，就写好了，其实总体还是花了很长时间（主要是在完成后全面的思考整个过程），我开始对进程切换中esp和ebp寄存器的值的变化很疑惑，通过打印输出的方式以及看反汇编代码，我成功找到了答案，esp值产生于在创建上下文时，context指针作为返回值将值传给了esp，而ebp的值是由调用函数的开头push ebp 和mov esp, ebp给予的，至此我就明白了整个过程。

第二个其实也不难，但是我犯了一个低级错误，是栈的生长方向弄反了，导致参数一直是0，后来才发现。

后面就是纯粹自己开始看讲义没彻底理解，对于用户栈和内核栈没有充分的认识。开始把内核栈当参数栈，没想到居然通过了，过了参数的框，发现不太对劲，雨水重新理解讲义，发现果然是自己的问题。再次好好看将以后，将几个关键位置调整后（往栈上放置参数时，我的代码是先设置了一个临时变量，所以后来不用重写，体会到了可移植性的好处），然后改指针，成功。

下一个麻烦是在实现带参数的execve，因为我在pa3还埋了一个坑，先填了坑才先弄好。此时出现bug我都能很快自己修好，通过调试理论应用和输出调试法，debug已经变得没有开始那么困难，现在，主要困难在于理解讲义和一些细节的实现。

后面的部分是在刚写完4.1后写的。

首先要吐槽，为什么2020 pa4开始比往届加了这么多内容，（先提前为下届及下下一届点根蜡(=v=)）。

好了，讲一讲busybox的感受，首先这次自己探索使用还是很有成就感的。通过自己对整个系统的理解，我很快就找到该如何在自己的程序上启动busybox，这使我十分开心，但是中间也发生了读入的参数是乱码的情况，后来调整了参数的解析方式才解决，在这里不能不敬佩linux内核以及shell工具的开发人员，仅仅一个参数解析就令我十分难受：泪目

现在感觉当对程序运行的整个过程有了充分的理解之后，找问题和解决问题都变得容易了很多，之前debug确实是有些慢。

PA 4.2

分页我感觉其实不是最难的，而PA3的SDL才是最花时间的。首先是map映射函数的书写，开始做这个我花了两个晚上，第一个晚上基本没有写代码，主要在看分页的原理以及i386手册，并上网查找资料了解整个过程，后来证明，我的这个决定是十分正确的，在后来，我基本没有犯过很重大的错误。（看手册和学习理论十分重要，经过前面无数的经验教学总结出来的）

写好了map，下面就是写硬件的相应配合了。这块起始也不是很好写，也做了一些copy and paste的问题，但是我很快意识到模块可以复用，于是整个思路就清晰了，自己写了一个page_table_walk函数，所有的操作基本都可以用到。

写好后，我在测试dummy的时候先补充了read cross page，确实很好写，还是源于可以复用的page table walk，同时还用了一些小trick，就成功解决了问题。不过在这里还是卡了一小会儿，主要是没好好看讲义，调度代码没改，意识到后很快就搞定了。

最麻烦的就是loader了，我在这里一共写了两遍，第一遍写得很丑，第二遍我采用宏定义的方式，使代码大幅度简化，不过这时还是犯了一个小错误，导致后面debug了一个晚上。不过这时，dummy已经可以正确运行了。

```
16  #define OFFSET 0xfff
17  #define page_begin_vaddr(A) ((void*)((uintptr_t)(A)&(~OFFSET)))
18  #define page_end_vaddr(A) ((void*)(((uintptr_t)(A)&(~OFFSET))+PGSIZE))
19  #define paddr_offset(A,B) ((void*)(((uintptr_t)(A)&(OFFSET))|(uintptr_t)(B)))
20  #define min(A,B) ((uintptr_t)(A) < (uintptr_t)(B)? (A):(B))
```

最艰难的就是pal的debug了，已知检测到present为0，而且出现的问题地方的值是一个随机的值，经过printf调试输出后，先定位到出问题的pc的位置，然后查看这个值从哪里来，但是这时就卡住了，觉得没有问题，但是确实出现了。

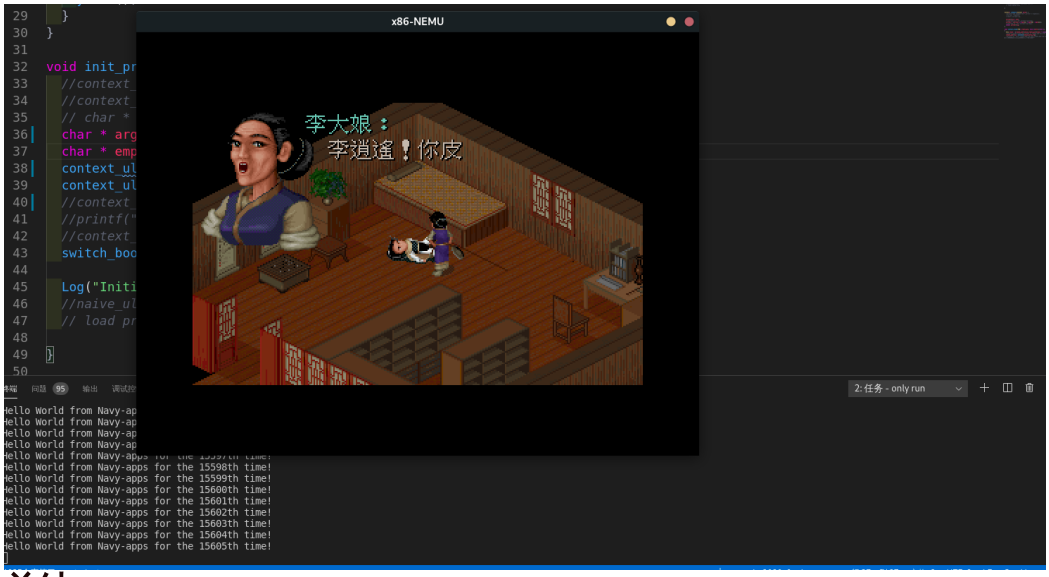
经过通过的调试过程，我终于意识到是loader给自己埋下了一个大坑，在分配物理页时，由于不一定是整数页，所以导致在memset时，实际上由于当前页已经映射，导致物理页中的数据并没有清零，然后出现了随机的错误。找到错误后，很快我就解决了问题。

最后的部分就很简单了。

PA 4.3

注入时钟中断难度不大，但是刚开始一直出现随机的错误，后来通过猜测，确定了错误位置，发现是操作数位数的问题，就解决了。

栈切换主要是看和理解的时间比较长，看了好长时间讲义和手册以及课本，明白了整个机制的运行过程（大概花了一整天），当明白了整个过程主要是硬件工作就吗，明白了。然后很快就写好了，最后的展示也很简单。



总结：

PA的收获还是很大的，对计算机系统有了更加深刻的认识。不过整体还是很痛苦，深陷BUG，看不到希望也有过几次，不过当做完后，还是有满满的成就感。pa4-3pa4-3

2020 PA 1

姓名:赵超懿 学号:191870271 匡亚明学院

实验进度:

我完成了所有实验内容

必答题

1.我选择的ISA是 x86 。

2.画出1+2+。。。+100的程序状态机 (PC , r1 , r2)

$$(0,x,x)->(1,0,x)->(2,0,0)->(3,0,1)->(4,1,1)->(2,1,1)$$
$$->(3,1,2)->(4,3,2)->(2,3,2)->(3,3,3)->(4,6,3)->(2,6,3)$$
$$->.....->(3,4851,99)->(4,4950,99)->(2,4950,99)$$
$$->(3,4950,100)->(4,5050,100)->(5,5050,100)->(5,5050,100)$$

3.理解基础设施

使用GDB进行调试时间:

$$t = 500 \times 90\% \times 30 \times 20 = 270000(s) = 75(h)$$

使用简易调试器

$$t = 500 \times 90\% \times 10 \times 20 = 90000(s) = 25(h)$$

节约50h

4.RTFM

x86:
EFLAGS寄存器中的CF位是什么意思?

Page 33 of 421 , 2.3.4 Flags Register 以及 Appendix C Page 419 of 421

ModR/M字节是什么?

Page 241 of 421 ,17.2.1 ModR/M and SIB Bytes 至 Page 245 of 412

mov指令的具体格式是怎么样的?

Page 345 of 421 , Mov -- Move Data 至 Page 351 of 421

shell命令

使用 find -name "*.[h|c]" | *xargs wc -l*

结果是 6024

去掉空行 find -name "*.[h|c]" | xargs grep -v "^$" | wc -l

结果是 5017

未编写前 分别为5322 4345

RTFM:

gcc -Wall 选项显示所有警告

gcc -Werror 将所有警告视为错误,并报错

作用:将所有警告视为错误报错,可以最大可能减少隐患和将来可能出现的错误。

实验感受：

这一部分是后来返回来写的。

总体来说，刚刚开始的PA就像一个新的世界，我第一次了解到可以这样写代码，内有不断有“还有这种操作“的感慨，大一写的代码感觉就是小打小闹。

第一阶段还是比较友好的，首先，总的东西不多，主要是表达式求值，难度也不是很大（和后面相比）。

第一个成就感来自于x86的寄存器的代码，通过自己差union和struct的匿名使用方法，了解到struct和union居然可以这样用（我之前甚至都不知道union，555），通过一晚上的努力，我终于完成了这个内容。

主要困难还是在于阅读那一段监视器的框架代码，不过难度也不大（x），主要是要找到对应的API，写的时候仿照框架代码就对了。

最后是调试表达式求值的部分，这一部分首先讲义有充分的提示，然后就是基本功的问题，（可惜我的基本功并不是很好）还是花了不少时间，不过由于问题集中，总能调出来。在这里我还没学会去找好用的工具，基本就是目力debug和make GBD，然后还没改O2的优化，（充分凸显刚开始的年幼无知.jpg）

从现在的观点来看，我觉得monitor后面的作用可能是没有其作为一个让我这种菜鸡从什么都不会到学会看框架代码，学会写一个稍微长一点代码的练习的作用大的，正是PA1的这一段的适应，让以后的路没有那么艰难。

PA1最大的帮助是让我接触一个新的世界，从什么都不知道到能做一点点，并且学会了很多东西的用法。从此开始了c语言的补课之路（x）。

PA2 实验报告

赵超懿 匡亚明学院 191870271

PA4 实验报告

实验进度：

必做题：（架构x86）

实验过程以及实验心得（包括一些我理解的蓝框题的回答）：

PA 4.1

PA 4.2

PA 4.3

总结：

2020 PA 1

姓名:赵超懿 学号:191870271 匡亚明学院

实验进度

我已完成PA2 **全部必做内容** 以及选做malloc实现并运行超级玛丽

必答题：

1.理解YEMU如何让执行程序

执行加法程序的状态机：

在执行加法指令时：不会改变内存的状态，只改变pc和寄存器的值。

(pc,R[0],R[1],R[2],R[3],内存) --> (pc+1, R[0]+R[1],R[1],R[2],R[3],内存)

pc加1指pc移动到下一个字节。初始R[0],R[1],R[2],R[3]的四个位置分别代表寄存器0，1，2，3，图中假设加法指令rt=0,rs= 1,则R[0]位置上的指变为R[0]+R[1]。若使用rs核rt来表示，即rt位置上寄存器的值变为R[rt]+R[rs]。加法指令不会改变内存上的值，故内存不会变化。

YEMU执行指令的过程：首先从存储的程序中取得指令，然后根据前4为来判断指令类型，然后对操作数进行译码，执行对应的操作，同时在译码过程中，pc指向下一条指令。

将程序看作状态机，其下一个状态既和当前的状态有关，也和输入即指令有关。加法指令是YEMU执行过程中一个特例。

比如mov指令 MOV reg32, imm32 ，其操作码为0xb8+rd，后跟有+rd，代表后面指令的译码结果是32位的寄存器,并根据rd得到目标寄存器，然后会有4字节的立即数。在nemu中，在exec_once中首先获取opcode位0xb8，switch语句定位到IDEX(0xb8+rd,ov_I2r,mov)，然后根据宏定义，先设置位宽位4字节（没有0x66开头的情况下），然后调用def_DHelper(mov_I2r)进行译码，mov_I2r的译码先调用decode_op_r译码得到寄存器的编号，然后处理好指针ddest指向目标寄存器。然后调用decode_op_I译码得到4位立即数，即第二个操作数，处理好指针idsrc1，然后为执行部分，调用def_EHelper（mov）实现指令。最后返回更新pc，进入下一次循环。

3.程序如何运行：理解打字小游戏如何运行。

答：main开始先进行ioe和video的初始化。

ioe_init检测设备是否实现，video_init对屏幕初始化，背景设为紫色，对字母设置颜色。在循环中，根据fps确定刷新频率，game_logic_updata(current)对现存的所有字符的位置进行刷新，刷新的新位置由当前位置核速度计算得到，对于超出的屏幕的字符，统计错的字符数量，使其停留一段时间。对于时间已到的字符，令其消失。再有一个while循换获取键盘的输入，判断是否退出游戏还是不改变还是使字符消失，check_hit函数，判断是否输入正确，若正确成功数量增加，使字符反向返回。最后的render对屏幕先刷新，再对字符的现在的位置进行染色，输出到端口，打印当前的正确核错误信息。

在整个运行过程中，打字小游戏在以x86为架构的nemu运行，使用x86架构的nemu计算出屏幕上每个位置的颜色，然后使用am的软件接口把数据输入到端口，然后nemu从接口获取信息将输出反馈在屏幕上

4.在`nemu/include/rtl/rtl.h`中, 你会看到由`static inline`开头定义的各种RTL指令函数. 选择其中一个函数, 分别尝试去掉`static`, 去掉`inline`或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

static inline都去掉：编译报错，原因在于重复定义，因为在exec.h中使用include包含了rtl.h,会使得同一个函数被多次定义（既在exec.h中，也在rtl.h中）

去掉static产生的结果也是重复定义函数，原因和上面的相同。

只去掉inline 编译器会发出警告，提示function defined but not used，但是由于Makefile中-Wall 和-Werror，会报错。在decode.c编译时会定义而不被使用。原因在于inline会把函数放到调用者内，而不是把函数放置等待调用，故不会出现函数被定义而不被使用的情况。

编译与链接

1. 在`nemu/include/common.h`中添加一行`volatile static int dummy;`然后重新编译NEMU. 请问重新编译后的NEMU含有多少个`dummy`变量的实体? 你是如何得到这个结果的?

   27个，在编译后得到的文件中使用grep搜索dummy的.o文件，共27个；

   shell grep -r -e 'dummy' |wc -l，这样得到是文件中含有dummy的文件的数量，如果在grep后加上-c，可以得到总数为54,但是由于dummy这样的定义为弱符号，每个文件中最后会随机挑选一个编译，故最终数量仍然为27个。
2. 添加上题中的代码后, 再在`nemu/include/debug.h`中添加一行`volatile static int dummy;`然后重新编译NEMU. 请问此时的NEMU含有多少个`dummy`变量的实体? 与上题中`dummy`变量实体数目进行比较, 并解释本题的结果.

27个，和第一问结果相同。原因在于common.h中包含了debug.h，所以当common.h被include时，debug.h也被include。

3. 修改添加的代码, 为两处`dummy`变量进行初始化:`volatile static int dummy = 0;` 然后重新编译NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)

   报错：重复定义。原因在于dug.h包含在common.h，这样会对dummy进行两次定义（初始化后即为强符号），所以报错。

了解Makefile 请描述你在`nemu/`目录下敲入`make` 后, `make`程序如何组织.c和.h文件, 最终生成可执行文件`nemu/build/$ISA-nemu`.

答：格式 file : file 1,file 2,... file 3表示文件file由后面的文件构成,**省略file文件本身**

然后省略了common.h（不直接include的）, 以及common.h include的debug.h 和macro.h.

在执行编译时,起作用的主要是这里

```
$(OBJ_DIR)/%.o: src/%.c
  @echo + CC $<
  @mkdir -p $(dir $@)
  @$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
```

一个.o文件由对应的.c文件及CFLAGS中的编译选项完成

```
INCLUDES  = $(addprefix -I, $(INC_DIR))
CFLAGS   += -O2 -MMD -Wall -Werror -ggdb3 $(INCLUDES) \
      -D__ENGINE_$(ENGINE)__ \
      -D__ISA__=$(ISA) -D__ISA_$(ISA)__ -D_ISA_H_=\"isa/$(ISA).h\"
```

可以看到 $(INCLUDES)给出了要生成.o文件需要的头文件的路径（标准库中的不需要写出），这样是使用gcc的过程中，编译器就可以找到对应的头文件，

`gcc g-O2 -MMD -Wall -Werror -ggdb3 -I./include -I./src/engine/interpreter g-gdgd -c -o build/obj-x86-interpreter/device/alarm.o */*.c`这样的格式，便可以生成.o文件，最后是文件的链接；

```
gcc -O2 -rdynamic  -o build/x86-nemu-interpreter
build/obj-x86-interpreter/device/alarm.o
build/obj-x86-interpreter/device/audio.o
.............(省略)
```

`build/obj-x86-interpreter/engine/interpreter/init.o -lSDL2 -lreadline -ldl`

链接过程将所有文件链接起来。

makefile中还提供gdb，和run操作，分别对应shell中的gdb和执行二进制文件

整个过程中，gcc -I -I./include -I./src/engine/interpreter,从这两个路径搜寻.h文件，以下是详细包含（直接include的，间接包含的会在-I后的路径去寻找）的文件

alarm.c : common.h time.h signal.h

audio.c: 未实现的情况下为 common.h

serial.c：map.h

keyboard.c: map.h monitor.h SDL.h

intr.c: isa.h

device.c: common.h

vga.c: comon.h

map.c: isa.h paddr.h vaddr.h map.h

mmio.c: map.h

port-io.c: map.h

monitor.c: isa.h paddr.h monitor.h getopt.h stdlib.h

log.c: common.h stdarg.h

ui.c: isa.h expr.h watchpoint.h stdlib.h readline.h history.h monitor.h paddr.h

expr.c: isa.h regex.h paddr.h

watchpoint.c: watchpoint.h expr.h malloc.h

cpu-exec.c: isa.h monitor.h difftest.h stdlib.h time.h watchpoint.h

dut.c: dlfcn.h isa.h paddr.h monitor.h

paddr.c: isa.h paddr.h vaddr.h map.h stdlib.h time.h

main.c: stdio.h stdlib.h string.h

reg.c: isa.h stdlib.h time.h reg.h

decode.c: exec.h rtl.h reg.h decode.h

logo.c: 无

special.c: exec.h monitor.h difftest.h

exec.c : exec.h decode.h

intr.c: exec.h rtl.h

mmu.c ； isa.g vaddr.h paddr.h

实验感受

PA2的这部分也是写完了才返回来写的。

首先是刚刚进入PA2时的感受，我整个人都不好了，看了一天的框架代码，也没完全搞懂整个运作机制（我太难了），只能继续看了，然后学习就这样痛苦了半周，才差不多弄明白nemu的运行。然后发现其实页不是很高深，就是自己的看框架代码的能力太弱了。

好了，可以通过cpu-test来写nemu的代码了，殊不知真正的痛苦才刚刚开始。（天真.jpg）在写代码的过程中，我逐渐体会到不严格按照手册的后果，一会挂这个一会挂另一个，这个人都傻了。（在写完test之前还没有看difftest，看到后是真的后悔没先看一遍PA2的讲义）

第一个调了很长时间才调好的是CF和OF，通过强制类型转换，我才解决了这个问题(之前写代码基本没用过，在PA中很快就学会了)。然后啪的一下，很快cpu test都写完了，但是有一两个错了，这便是第二个问题，nemu中的规约使用出问题了，这个问题比较隐蔽，是另一个rtl函数调用了sext函数（也是rtl），然后两个函数都使用了t0寄存器，直接挂掉，总之，PA2中的问题都可以归结为RTFM，确实不好好按照规约来写害人不浅。

在写完指令后我就到了快了的环节，首先是快乐的跑分环节，当然在跑分过程中我借助difftest找到很多bug，比如指令位宽等等，不过解决起来还是很快的（体会到了真机的好处，可以跑KVM并且性能极佳，difftest可以到50W条指令每秒，比qemu好太多）然后就是没事跑一跑分，实现了malloc打一打超级马里奥。然后就到了PA2.3

2.3中时钟就卡了好久，最后**感谢贾林杰同学**的提示，实现时钟，后面的就很好解决了，不过中间还有一个错误提醒了我这是一个小端机，在以后的pa中我时刻不敢忘（这个错误出现在VGA读取屏幕长和宽时，由于小端机刚好读反，最后通过打印调试法才解决，在之后的PA3和PA4涉及需要自己读取内存时，时刻注意到这是一个小端机）然后PA2就顺利结束了。

总结

PA2中在写指令时，我深陷bug之中，那是真的感到绝望，不过很庆幸，最后我凭借自己走了出来，在这其中，我深刻体会手册的重要性，也十分佩服编写手册的人，居然可以让如此复杂的计算机精密运转而不出错（tql）。

然后是在PA2中学到的知识，首先明白了指令执行的整个过程，然后学会了基本的调试方法，对整个计算机系统的硬件层有了更加深刻的认识。

PA3 实验报告

匡亚明学院 赵超懿 191870271

实验进度：

我完成了PA3中所有必做内容，关于蓝框选做，我仅完成了使画布位于屏幕中央。

必做题：

1. 理解上下文结构体的前世今生 (见PA3.1阶段)

   答：这个问题我再解决时主要从指令的执行角度来解决。

在保存上下文的过程中，cpu执行了int 指令后，进入到了raise_intr函数，在这个函数中，cpu按照顺序，先后将eflags，cs，ret_addr压入栈中，然后通过idtr寄存器计算出跳转的地址，然后下面跳转到__ am_vectrap函数，向栈中压入 0x81(int指令的值) _am_asm_trap函数，在这个函数中，调用了pusha和push 0，然后是esp寄存器，然后进入到call __am_irq_handle函数，将使用栈存储的数据，即context。

所以在context结构体中的，在栈中从高地址到低地址依次为 eflags, cs ,ret_addr , 0x81,所有通用寄存器以及0和esp，所以只需要在**Context**结构体中将以上**对应的值倒过来按顺序**写在Context中即可。在这个结构体中，void *cr3 指针应该指向push的0，其余对应相应的值。

对于是如何组织起来的，在于数据在栈上的组织，最后push进入的esp是作为传入的Context*指针参数，其指向的结构体实体便是在栈上的压入的数据。指针参数于是传入了__am_irq_handle 函数。

2. 理解穿越时空的旅程 (见PA3.1阶段)

   答：yield()函数调用后，会执行int 81指令，然后按照在第一个必做题中所说：

   进入到了raise_intr函数，在这个函数中，cpu按照顺序，先后将eflags，cs，ret_addr压入栈中，然后通过idtr寄存器计算出跳转的地址，然后下面跳转到__ am_vectrap函数，向栈中压入 0x81(int指令的值) _am_asm_trap函数，在这个函数中，调用了pusha和push 0，然后是esp寄存器，然后进入到call __am_irq_handle函数。

   在__am_irq_handle 函数中，然后识别异常号81，将事件赋值为EVENT_YIELD, 然后进行事件处理，根据在cte_init中注册的事件来进行do_event进行事件分发，事件分发识别之前赋值的事件，判断事件为EVENT_YIELD后，输出一段话，然后开始返回，恢复上下文，程序返回到 _am_asm_trap 调用 _am_irq_handle之后的敌方，将栈指针esp增加到 pusha指令执行后的地方，然后将之前保存的通用寄存器全部pop到寄存器中，然后执行iret指令，将之前push进栈中的pc，cs寄存器以及eflags ，都pop到对应的寄存器中中，然后在yield()函数调用int 81后的位置返回，恢复到程序调用yield前的状态。

3. hello程序是什么, 它从而何来, 要到哪里去 (见PA3.2阶段)

   我们知道`navy-apps/tests/hello/hello.c`只是一个C源文件, 它会被编译链接成一个ELF文件. 那么, hello程序一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 它的第一条指令在哪里? 究竟是怎么执行到它的第一条指令的? hello程序在不断地打印字符串, 每一个字符又是经历了什么才会最终出现在终端上?

   答：hello程序被编译后，其ELF文件处于二进制文件的开头。Hello文件处于ramdisk.img文件的开头。

   nanos-lite中的init_proc函数调用naive_uload函数加载文件。最后调用loader.c文件加载文件。首先将大小为sizeof(Elf_Ehdr)的文件读入，根据elf头文件的格式，可以获得program header的数量，是否加载以及在文件中偏移量，于是loader.c根据这些信息读取program header文件偏移量，文件大小，内存大小，加载信息，虚拟地址，将文件读入到nemu的内存之中，而且elf header中有一个程序进入地址，e_entry，这样系统便可以将文件pc设置为loader的返回值，然后跳转，体现在指令上变为call *%ebx，于是就进入到了文件中的_start的位置（因为elf文件的e_entry指向这个位置），于是nemu开始执行客户程序hello。

   hello的执行过程中，首先调用的是write函数，先后调用 *write_r和 _write函数， _write函数即为libos中的 _write函数，开始系统调用，准备好参数，然后执行int 80指令，该指令属于系统调用事件，于是nanos-lite开始系统调用。这个过程体现在c的代码上为从navy-apps中的syscall.c文件中 _write 调用 _syscall*函数，从这里执行int 80 指令开始系统调用，系统调用识别为sys_write调用，然后交与函数处理，由于输出是stdout ，所以直接使用串口将字符输出。

之后的printf输出，（此处解释使用了_sbrk的情况），printf申请内存作为缓冲区，直到遇到\n输出。在要输出的过程中，库函数将输出行为转换为系统调用（到write），类似于write的行为，仍然使用fd为stdout的为sys_write的系统调用，使用串口将文件输出出来。然后循环这个过程。

4. 仙剑奇侠传究竟如何运行 运行仙剑奇侠传时会播放启动动画, 动画中仙鹤在群山中飞过. 这一动画是通过`navy-apps/apps/pal/repo/src/main.c`中的`PAL_SplashScreen()`函数播放的. 阅读这一函数, 可以得知仙鹤的像素信息存放在数据文件`mgo.mkf`中. 请回答以下问题: 库函数, libos, Nanos-lite, AM, NEMU是如何相互协助, 来帮助仙剑奇侠传的代码从`mgo.mkf`文件中读出仙鹤的像素信息, 并且更新到屏幕上? 换一种PA的经典问法: 这个过程究竟经历了些什么?

   答：

```
166     //
167     gpGlobals->f.fpFBP = UTIL_OpenRequiredFile("fbp.mkf");
168     gpGlobals->f.fpMGO = UTIL_OpenRequiredFile("mgo.mkf");      Zihao
169     gpGlobals->f.fpBALL = UTIL_OpenRequiredFile("ball.mkf");
170     gpGlobals->f.fpDATA = UTIL_OpenRequiredFile("data.mkf");
171     gpGlobals->f.fpF = UTIL_OpenRequiredFile("f.mkf");
172     gpGlobals->f.fpFIRE = UTIL_OpenRequiredFile("fire.mkf");
173     gpGlobals->f.fpRGM = UTIL_OpenRequiredFile("rgm.mkf");
174     gpGlobals->f.fpSSS = UTIL_OpenRequiredFile("sss.mkf");
175
```

   从这里加载文件，之间有很多定义的函数，最终会到这里使用fread函数

```
558   //
559   FILE *fp = UTIL_OpenFileAtPath(gConfig.pszSavePath, PAL_va(1, "%d.rpg", iSaveSlot));
560   //
561   // Read all data from the file and close.
562   //
563   size_t n = fp ? fread(s, 1, size, fp) : 0;      Zihao Yu, 2 months ago • common: merge som
564
```

fread是库函数，会产生系统调用SYS_read，这样库函数就和nanos-lite交互，系统调用会使用我所写的fs_read读取文件，最后会成功将数据读取出来。这使用系统调用读取数据的过程中，会使用AM提供的基本环境，如memcpy等，最后这些行为都被编译成为指令，在nemu上运行，这个系统在读取数据的过程中达到了协同。

在pal显示动画的过程中，

```
PAL_ProcessEvent();
dwTime = SDL_GetTicks() - dwBeginTime;

//
// Set the palette
//
if (dwTime < 15000)
{
    for (i = 0; i < 256; i++)
    {
        rgCurrentPalette[i].r = (BYTE)(palette[i].r * dwTime /
        rgCurrentPalette[i].g = (BYTE)(palette[i].g * dwTime /
        rgCurrentPalette[i].b = (BYTE)(palette[i].b * dwTime /
    }
}

VIDEO_SetPalette(rgCurrentPalette);
VIDEO_UpdateSurfacePalette(lpBitmapDown);
VIDEO_UpdateSurfacePalette(lpBitmapUp);

//      Zihao Yu, 2 months ago • common: merge some macros i
```

pal使用了SDL库中的函数（VIDEO_其实是SDL库函数）这些函数会使用在libs中miniSDL中的完成的SDL_Update ,SetPalette等函数，而这些函数又是通过NDL函数来实现，NDL中的函数又使用了fwrite等，触发**系统调用**，**在系统调用中通过AM提供的屏幕，键盘，时钟接口(io接口)（io_write以及io_read）来将画面等信息传递给硬件以及从硬件获取信息，**于是AM，nanos以及AM就协同了起来，最后这些用户程序运行在硬件nemu上，完成开始动画的显示。

实验过程以及实验心得（包括一些我理解的蓝框题的回答）：

总体体会：

在做pa3的3.1和3.2时，觉得难度一般，以为后面的pa大概就这样可以轻松的完成，但是pa3.3给了我致命一击。太难了！

PA 3.1

首先是pa3.1的完成过程：按照讲义实现寄存器，看手册实现行为，然后卡在了 上下文结构体。

上下文结构体（蓝框）：诡异的x86代码：我认为push 的esp的作为事件分发函数的输入参数，指向的是之前压栈的数据。

在这里我犯了一个很傻的错误，没注意到x86要去除一个push 0，于是发现怎么都不对。（！！！），其实这块也不难，就是找到栈中间中值的对应顺序，我看到这块就直到要这样做，但是没看到要改框架代码，于是白费两小时。

PA 3.2

然后是pa3.2的完成过程：首先是loader.c，开始就卡住了，发现加载的文件的大小都是0，（中间明白了二进制文件的复制（一定不是打开vim复制=:=））于是卡住好长时间了（看了makefile明白了update的作用，为以后也节约了一点时间）。最后是请教李晗同学解决了问题，在此感谢李晗同学的帮助。在李晗同学的指导下，我一点点查看编译信息，最后找到真相，是make的过程中需要resouce.S文件被更新，才能够是ramdisk.img文件被编译进去。在这个过程中，我对make的机制有了更加细致的理解。最后再次感谢李晗同学。

蓝框的回答：为什么要置0：这一块是用于全局变量和静态变量的初始化（开始我写的memcpy，后面发现不对才明白的（因祸得福！））

解决了loader，加载文件还是比较简单的，man elf，直接就按照要求将程序段加载。中间使用printf调试，很容易就能解决问题。在loader之后，直至pa3.2结束，都是比较顺利（指没有大问题）的，小问题还是很多的。

在完成系统调用时还是遇到一点问题的，主要是返回值的问题，比如忘了设置返回值，出现和讲义描述现象不相符的情况。不过最后还是很快解决了。最后的堆区管理，实现起来也是很简单，也就不贴代码了。

3.2的最后是堆区管理，**重点：这里没有遇到问题，后面出现问题后找来找去还是找到了这里，然后发现又不是这里的问题（太折磨了！！！）**

当然在3.2阶段还有最大的收获：**在这个阶段写系统调用的过程中，我突然就对整个PA有了很清晰的认识，从硬件到AM到naons以及navy，对于其整个大的工作机制基本都清晰了（之前一直有些模糊，不是很明白，现在总算是明白了整个系统的工作原理了！！！）**

PA 3.3 !!!

PA3的噩梦，感觉整个人变成了GDB。每天都在debug。基本要写的代码都重写过两次（基本全部重写），除了SDL_Pollevent和SDL_Waitevent和文件系统代码。（哭了：满怀信心开始pa3，结果整个人都裂开）

首先是文件系统：这个其实还好，查一查手册，了解下行为。基本就解决了，然后是重写loader，这个过程按理说应该是比较简单的，但是我不直到为什么就搞不对，留下了非常多的调试信息：

```
static uintptr_t loader(PCB *pcb, const char *filename) {
  //TODO();
  //uint32_t size = get_ramdisk_size();
  //printf("%d\n",size);
  Elf_Ehdr elfhdr;
  Elf_Phdr prohdr;
  //printf("%s\n",filename);
  size_t fd = fs_open(filename,0,0);
  //printf("%d\n",fd);
  fs_read(fd,&elfhdr,sizeof(Elf_Ehdr));
  //printf("%x %x\n",elfhdr.e_phoff,elfhdr.e_phnum);
  assert(fd != -1);
  //printf("%x %x %x\n",elfhdr.e_entry,elfhdr.e_phentsize,elfhdr.e_ehsize);
  for(int i = 0;i < elfhdr.e_phnum;i++)
  {
    fs_lseek(fd,elfhdr.e_phoff+i*sizeof(Elf_Phdr),SEEK_SET);
    fs_read(fd,&prohdr,sizeof(Elf_Phdr));
    if(prohdr.p_type == PT_LOAD){
      fs_lseek(fd,prohdr.p_offset,SEEK_SET);
      fs_read(fd,(void *)prohdr.p_vaddr,prohdr.p_filesz);
      //printf("from %x %x size = %x \n",prohdr.p_vaddr,prohdr.p_offset,prohdr.p_filesz);
      memset((void*)(prohdr.p_vaddr+prohdr.p_filesz),0,prohdr.p_memsz-prohdr.p_filesz);
    }//之前用的memcpy，不愧是我

  }
  fd = fs_close(fd);
  assert(fd == 0);
  printf("%s File Loaded\n",filename);
  return elfhdr.e_entry;
}
//静态变量的初始化问题，好像是在programmmer header中加载的，
```

可以看到有很多调试信息，后面的文件基本也都是这样过去的。（调试的代码基本和代码差不多一样了，这还是删掉了一些的情况下，不过调试理论还是很好用的）。

操作系统上的IOE，如果在PA2没有理解整个PA的工作机制，那么我可能还要花时间去理解，但是在这里，我已经明悟了，所以写起来很快（仅仅是写555），串口和时钟还是比较简单的，然后在仅在此处是屏幕正常工作也是比较简单的。对的，仅在此处，这也是为什么我后来改了很多次的原因。

蓝框选做：在理解画布概念后实现，不过在第一次写并没有写。

下面就到了错误时刻：（先跳过顶点算数，后面再讲，快进到APP）

1. nslider，一次成功：为什么再第一次实现有问题的基础上可以对呢？因为恰好全屏（555）
2. menu 也是一次成功：成功原因同上：全屏
3. nterm是第一个转折点：首先：在实现NDL时屏幕我第一次用了sprintf，在用这个的时候我还考虑到要以%c来进行输出。在第一次运行nterm时，我发现结果是满屏的白色，然后我阅读了源码，发现结果不应该是满屏，然后就修改了opencanvas函数，修改好后，可以正常显示了，然后发现帧率过低，然后开启了第一次重构，将sprintf修改为fwrite，然后发现性能大幅度提升。

第一阶段到此结束，似乎还挺顺利。（我当时也这么觉得），然后后来就不是了，当开始运行仙剑奇侠传时，一切埋下的坑就出现了（不仅仅是软件，还有硬件，我裂开来）

第二阶段：首先是nanos的问题，首先，由于对手册理解不正确，pal直接黑屏，然后我去看自己的源码，发现对画布的理解有问题，然后从NDL_Draw到SDL基本和画布相关的都重新写了一遍，然后先重复之前的测试，然后在前面的就直接挂掉，好，我继续bug，再次走到pal面前，此时，我的video实现基本都还可以（还是有一些小问题的），然后出现了非常神奇的错误，PAL的启动画面能否成功变成随机的，想了一段时间意识到和随机有关系的只有时间了，啪的一查，很快啊，果然是时间错了（除法变成了取模），我看着我之前犯下的

xx错误，觉得我是个x（不是）。好了现在可以正常启动pal，果然不再是随机事件了。第二阶段结束。

然后就到了第三阶段：进去，选项上面有黑色阴影，此时我甚至想着是不是因为我们使用了RGB而没有用a的值导致的，事实证明，机器永远是对的，只要有错那么一定是我（太真实了），为了这个错误，我阅读了很多仙剑的源码（看这种规模的文件好痛苦，而且重点还有我的vscode有些好用的功能无法对pal生效，我又裂开）。然后慢慢看源码，然后打断点，输出调试，一上午过去才修好这个bug，果然还是写SDL对画布的理解不够，好，再次重写（小规模重写），修好了我都要泪目了，然后x86跑有些，进去就发现两个问题，首先打不了怪，然后一遇到有任务对话，直接错误，我直接懵逼(55555555)，然后调了很长时间，没有进展。后来还发现了堆区一直增长的问题，（上面提到过，再次裂开）

我周日一天就做了这么一点事。当然，还学会了用navy作为基础设施。但是还没有彻底定位到错误（那天晚上才学会用navy作为基础设施的）。然后我偶然发现了一些问题，堆区还在不停增长——sbrk出问题了，但是还是不会解决。

终于要到结局了：（太难了）

首先是堆区问题，当发现即使不动程序的情况下，他仍然在增长，然后看了1个多小时没想法，只发现了是fopen在不停申请，然后我去洗澡，在洗澡时我突然明悟了NDL_init的作用，就是设置一些全局变量的指针，在init时对其初始化，这样就不会一直打开了（Yes！）

然后是pal运行遇到对话直接退出的问题，我在学会了native这个强而有力的工具后很快确信是我的硬件层nemu出bug了，我就很绝望，然后想怎么办呢？先往下看，看到difftest的开关，有希望，然后上手一做，各种出问题（有些难），然后就卡住了。最后是在方宇航同学的帮助下进行替换测试（仅nemu），终于定位到了nemu的错误是eflags的设置，然后解决，进行pa2中进行的一系列测试，然后跑跑仙剑，大功告成（完结撒花）。

收获：首先是实验课刚讲的调试理论，有了很大的作用，我打端点的能力大幅度提升，在后期深陷bug中时也能很快找到bug，然后是替换测试（也是上课讲的），同样使用了类似调试理论的方法，二分快速定位到bug。

感受：

机器真的永远是对的！！！

即使经过充分的测试，依然很有可能出现bug，在我确信是我的nemu'出现问题是，我很难相信，因为在pa2中，我的nemu可以运行讲义要求的所有指令，并且带difftest依然可以通过所有的测试。所以，经过测试的代码依然要**谨慎对待**。

关于测试，定点数也是需要测试的（5555），之前写过bug导致pal战斗不正常。也验证了为经过测试的代码都是不正确的。

对以后的启示：在写pa3的过程中我的方法确实不好，经常出现对手册理解不到位，观点不全面的原因，在SDL中体现的更加明显，写有一定规模的代码一定**要从全局出发，规划好理解到位再写（当然边看边写容易增加理解）**。另一个方面就是工具了，vscode的使用在pa3中给了我很大的帮助（之前花了一些时间在学习vscode是值得的。）

最后，我也很高兴，在最后深陷bug长达1周的时间中，我没有放弃。在pa3的整个过程中，除了两位同学的帮助，我的其他内容均独立完成，自己查看手册，自己找bug，自己修好，写好后又重构，一遍又一遍，体会到的写一份好的代码的困难，**在这个过程中，我的能力也得到大幅度提升。**回首pa3，还是一份愉快的旅程。

实验报告在pa3仙剑奇侠传成功运行后完成，有很多细节遗漏，以后应该注意，应该及时记录问题，跟着pa的进度写实验报告。

最后的截图的展示：

注释与代码的交错（写了很多遍，也调试了很多），纪念不易的PA3。

```
void SDL_SetPalette(SDL_Surface *s, int flags, SDL_Color *colors, int firstcolor, int ncolors) {
  assert(s);
  assert(s->format);
  assert(s->format->palette);
  assert(firstcolor == 0);

  s->format->palette->ncolors = ncolors;
  memcpy(s->format->palette->colors, colors, sizeof(SDL_Color) * ncolors);
  /* for(int i = 0;i < ncolors;i++)
    printf("%dth r = %x g = %x b = %x\n",i,colors[i].r,colors[i].g,colors[i].b); */
  //printf("%d\n",s->flags);
  if(s->flags & SDL_HWSURFACE) {
    assert(ncolors == 256);
    /* for(int i = 0;i < ncolors;i++)
      printf("%dth r = %x g = %x b = %x\n",i,colors[i].r,colors[i].g,colors[i].b); */
    /*printf("\nnew\n");
    for(int i = 0;i < s->w;i++)
      printf("%dth  color = %x\n",i,colors[s->pixels[i]]); */
    for (int i = 0; i < ncolors; i ++) {
      uint8_t r = colors[i].r;
      uint8_t g = colors[i].g;
      uint8_t b = colors[i].b;
      //printf("ith = %d r = %x g = %x b = %x\n",r,g,b);
    }
    /* for(int i = 0; i< s->w*s->h;i ++)
      printf("place %d   idx = %d color = %x\n",i,s->pixels[i],colors[s->pixels[i]]); */
    SDL_UpdateRect(s, 0, 0, 0, 0);
    //while(1);S      You, seconds ago • Uncommitted changes
  }
```

```
  uint8_t* dst_color = dst->pixels,*src_color = src->pixels;
  uint32_t color_width = dst->format->palette?1:4;
  //printf("color width %d\n",color_width);
  for(int i = 0;i < src_h;i++)
    memcpy(dst_color+color_width*((i+dst_y)*dst->w+dst_x),src_color+color_width*((i+src_y)*src->w+src_x),color_width*src_w);
    //memcpy(dst_color+4*(i+dst_y)*dst->w+4*dst_x,src_color+4*(i+src_y)*src->w+4*src_x,4*src_w);
  /* for(int i = 0; i < h;i++)
    //memcpy(dst_color+dst_w*(y+i)+x,src_color+i*w,4*w);
    for(int j = 0;j < w;j ++)
    {
      dst_color[dst_w*(y+i)+x+j] = src_color[i*w+j];
    } */

  //printf("please implement me\n");
  //assert(0);
}
```

```
  for(int i = 0;i < h;i ++)
    for(int j = 0;j < w;j ++)
    {
      canvas[(y+i)*canvas_w+x+j] = pixels[i*w+j];
    }
  for(int i = 0;i < canvas_h;i ++)
  {
    //printf("seek %d\n",4*((i+place_y)*screen_w+place_x));
    fseek(fb,4*((i+place_y)*screen_w+place_x),SEEK_SET);
    fwrite((void*)(canvas+i*canvas_w),1,4*canvas_w,fb);
  }
  //printf("NDL %p %d\n",fb_sync,ftell(fb_sync));
  fseek(fb_sync,0,SEEK_SET);
  fwrite("1",1,1,fb_sync);
  //printf("refresh %d\n",k++);
  /* for(int i = 0;i < h;i++)
  {
    fseek(fp,4*((y+i)*screen_w+y),SEEK_SET);
    fwrite((void*)pixels,sizeof(uint32_t),w,fp);
    //printf("seek %d\n",4*((y+i)*wid+y));
    for(int j = 0;j < 4*w;j++)
    {
      fwrite((void*)pixels,sizeof(uint32_t),h,fp);
      fprintf(fp,"%c",((char*)pixels)[4*w*i+j]);
      //printf("p = %d %d\n",w*i+j,pixels[i*w+j]);
    }
  }*/
  /* printf("%p %d\n",fb_sync,ftell(fb_sync));
  fseek(fb_sync,0,SEEK_SET);
  sprintf(fb_sync,"%c",'1'); */
  //fwrite(buf,1,1,fb_sync);
```

```
  Finfo* f = &file_table[fd];
  if(f->read == NULL)
  {
    if(f->open_offset >= f->size)
    {
      assert(f->open_offset <= f->size);
      //printf("%s open_offset = %d size = %d FULL\n",f->name,f->open_offset,f->size);
      return 0;
    }
    else {                                                  size_t <unnamed>::open_offset
      size_t l = len <= file_table[fd].size - file_table[fd].open_offset? len:file_table[fd].size - file_table[fd].open_offset;
      //printf("third %d %d %d\n",file_table[fd].open_offset,file_table[fd].size,l);
      ramdisk_read(buf,file_table[fd].disk_offset+file_table[fd].open_offset,l);
      file_table[fd].open_offset = file_table[fd].open_offset+l;
      assert(file_table[fd].open_offset <= file_table[fd].size);
      return l;
    }
  }
  else{
    //printf("%d\n",len);
    int ret = f->read(buf,file_table[fd].open_offset,len);
    f->open_offset+=len;
    //size_t l = len <= file_table[fd].size - file_table[fd].open_offset? len:file_table[fd].size - file_table[fd].open_offset;
    //file_table[fd].open_offset = file_table[fd].open_offset+l;
    return ret;
  }//以后可能出现问题

}//remained to be thinking about fd == 0,1,2, .e.t stdin,stdout,stderror
  // else if(file_table[fd].open_offset+len > file_table[fd].size)
  // {
  //   printf("sec2 %d %d %d\n",file_table[fd].open_offset,file_table[fd].size,len);
  //   ramdisk_read(buf,file_table[fd].disk_offset+file_table[fd].open_offset,file_table[fd].size-file_table[fd].open_offset);
  //   file_table[fd].open_offset = file_table[fd].size;
  //   printf("%d\n",);
  //   return file_table[fd].size-file_table[fd].open_offset;
  // }
```

```
  if(f->write!=NULL)
  {
    //printf("%s %d\n",f->name,f->open_offset);      tracer-ics2020, 14 hours ago • >  run
    //printf("size = %d offset = %d len = %d\n",f->size,f->open_offset,len);
    int l = f->write(buf,f->open_offset,len);
    //printf("buf = %x\n",*(uint32_t*)buf);
    //printf("fd = %d open = %d\n",fd,f->open_offset);
    f->open_offset += l;
    return l;
  }
  else{
    int ret = file_table[fd].size - file_table[fd].open_offset <= len? file_table[fd].size - file_table[fd].open_offset:len;
    ramdisk_write(buf,file_table[fd].disk_offset+file_table[fd].open_offset,ret);
    file_table[fd].open_offset +=ret;
    assert(file_table[fd].open_offset <= file_table[fd].size);
    //printf("write %s\n",f->name);
    return ret;
  }
  /* if(fd == 1||fd == 2)
  {
    for(int i = 0; i < len;i ++) {
      putch(((char*)buf)[i]);
    }
    //putch('\n');
    return len;
  }
  else if(fd == 0) return -1;
  else {
    int ret = file_table[fd].size - file_table[fd].open_offset <= len? file_table[fd].size - file_table[fd].open_offset:len;
    ramdisk_write(buf,file_table[fd].disk_offset+file_table[fd].open_offset,len);
    file_table[fd].open_offset +=ret;
    //printf("%d + %d %d\n",file_table[fd].open_offset,len,file_table[fd].size);
    assert(file_table[fd].open_offset+len <= file_table[fd].size);
    return len;
  } */
}
```

感谢

感谢方宇航同学在我在运行pal时解决nemu硬件 bug中提供的帮助。

感谢李晗同学的答疑解惑。

衷心感谢两位同学！