PA2.5

1. 理解mainargs参数是如何传递给hello程序的:
   1. 首先要明确，这个hello程序是要被nemu加载的程序，因此在hello被编译的时候，就需要把这个参数传递给hello里面的main函数了。因此我们可以定位这个参数是在编译的时候。但是目前只能找到makefile中关于把这个参数作为预定义的宏传给了hello.c函数。那么hello是如何根据这个宏来得到参数的呢？？？这个地方不理解！
   2. 补充a) 找到了这个宏在哪里被引用了！！！在abstract-machine/am/src/platform/nemu/trm.c这个函数下面！这个trm.c里面的内容还是非常重要的。他直接定义了halt、heap、以及trm的启动函数\_trm\_init。在\_trm\_init函数中，调用了宏mainargs并传递给了main函数！

另外，在start.S汇编文件中，定义了程序的入口\_start，并且从\_start启动进入\_trm\_init。

1. 在时钟update这一块的代码似乎是有BUG的！只有把高低位都拼接起来才能正确让us读出时间，但是只读地位却不行，us一直是0.这就非常奇怪了，因为地位肯定有数的呀！！所以可能是我的低32位数据的指令有BUG？？
   1. 进一步探索发现，只要把RTC\_ADDR的高位和地位都读出来就没有这个BUG。。。。。莫不着头脑～～。
   2. !!!原来这就是框架代码埋下的大坑！！！！他是通过寄存器来模拟MMIO的，所以他需要通过回调函数来更新，而回调函数只有在你读时钟外设高地址的时候才会更新。所以读取顺序要从高到低。。。。。。太坑了！！！

进入到ysyx基础阶段 B线

1. 又遇到了ALU的问题。重新思考了以下关于无符号数比较的问题。觉得可以类比一个12小时制的时钟。利用减法来比较时，对一个点取反相当于他在时钟上对称了一下，即时钟针到0/12点的距离在之前和之后都是一样的。这就像一个头尾连在一起的一个结构。因此无符号数利用减法比较时，当结果的carry有进位的时候，那么就说明他们相加溢出了！溢出就说明被减的那个时钟针在减数的后面或者相等！！非常巧妙好理解，尝试在纸上画一个12小时制的时钟就明白了。
2. 对于补码运算的符号溢出问题。 加法减法都转换为加法运算，而且数据以补码的形式进行运算，得出的结果就是补码形式，不论以有符号数还是无符号数的角度看，计算结果都是正确的！ 对于加法来说，只有正数+正数，负数+负数这两种情况会发生溢出问题，因此只需要特殊处理这两类。
3. 比较运算这个地方感觉实验讲义上有点麻烦。在不溢出的情况下，直接看符号为判断；在溢出的情况下，说明A，B肯定不同号，这个时候直接判断A、B的符号位不就行了吗？ 但是判断结果的符号位确实更省逻辑资源一些： 那么这样只剩两种情况： 正-负 ->正+正溢出，那么结果符号位肯定是1,这时候A为正， A>B;另一种情形就是负-正 ->负+负溢出，那么结果符号位肯定是0,这时候A<B；

A线

1. PA3.0
2. nanos的代码和NEMU的代码是分离的，互不影响。Nanos运行在AM之上
3. AM给的栈空间有多大呢？ 这一点应该根据连接器的脚本可以查询到。在AM目录下的link.ld文件查询，他在链接的时候通过-T被引用，可以看到栈空间的大小是0x8000，并且是0x1000字节对齐的。



8. PA3.1
9. 这一部分简直就是之前学习过的汪老师教的内容：异常上下文的保存已经跳转处理。总体的思路是这样的：软件通过ecall除法异常，这时候需要保存相应的寄存器，如mstatus，并把当前指针写入mepc中，然后跳转到mtvec，即中断向量地址，这个一般都是由操作系统控制的，也即操作系统此时获得控制权来处理异常。首先保存上下文，包括gpr，及一些csr到栈里，然后跳转到相应的中断处理函数处理中断，然后返回。最后从栈中恢复上下文，使用mret指令返回保存在mepc中的地址。其中有个问题，mepc的地址+4是软件控制的，因此需要操作系统自己判断什么时候+4，比如异常的时候不加，中断的时候+4.
10. 因此实现上述功能，需要添加和寄存器操作相关的指令，目前只需要csrrw、csrrs、mret这三个
11. 重新组织结构体比较容易，只需要看汇编里面的数据保存顺序就行了。因此另一个问题迎刃而解，上下文指针c是从汇编代码里面来的，由于RV的规范，参数传递都使用a0开始的寄存器，所以a0就是c，而a0指向的sp，sp指向的是刚才保存数据的栈顶，那么显而易见，他就是上下文结构体的指针，因此C里面都是上下文的数据。
12. 为什么RISC指令中mepc+4与否是由软件来控制，而CICS指令都是由硬件控制的？我的理解是减少硬件复杂度。软件可以自己判断来的是异常还是中断等。比如发生了分页错误等，那么处理完之后还需要返回原指令再检查看是否还出错，而中断的话，再处理完之后就可以直接跳到下条语句运行。那么如果把异常、中断的判断机制交给硬件，会需要更大的芯片面积和复杂的逻辑。
13. 必答题(需要在实验报告中回答) - 理解穿越时空的旅程
14. 解答如下：yield（）会触发ecall指令，那么NEMU中就会运行ecall，触发异常机制，会把相应的触发号等写入相应csr中，然后跳转到mtvec保存的异常向量表地址。这个地址在AM中被设置为\_\_am\_asm\_trap（）这个函数的地址，因此之后会保存上下文，跳转到中断处理函数，他会先识别出中断号，然后交给操作系统来处理。处理完成后返回一个上下文的结构体，（我的理解是为了给以后多程序运行切换使用的），然后回到trap里面恢复上下文，此时这个上下文有可能是另一个程序的哦！
15. PA3.2
16. 踩坑踩坑！！！此处img文件的program段的第一个可load的项，他的offset竟然是0！！看了很多遍手册，发现没有问题，就是从文件头开始的！！那么只有一个可能，就是程序的起始地址另有地方！！果然，是e\_entry!!
17. 在C语言中，## 是一种特殊的预处理器运算符，用于在宏定义中连接两个标识符或标记（tokens）
18. 必答题(需要在实验报告中回答) - hello程序是什么, 它从而何来, 要到哪里去
19. 这个问题我想已经很好回答了！hello程序就是一段被操作系统加载并执行的二进制文件数据。他由navy程序编译而来，在编译时由-T选项都被链接到0x83000000的地址，并且编译链接后得到相应的ELF文件。该文件里存放了所有的信息，比如可执行文件的开始地址，program header，section header信息等。之后os操作系统的程序被编译链接，在链接过程中，会把ramdisk，也就是hello程序的ELF文件数据都链接到ramdisk\_start到ramdisk\_end这段内存中。之后我们会首先开始运行操作系统，然后解析ramdisk的信息，并把需要加载的数据加载到0x8300000这个起始地址，并通过program entry地址找到hello程序的第一条指令地址。那么操作系统就可以跳转到这个地址运行hello程序拉！至于打印字符串这个行为，就是一个系统调用过程而以。
20. 又一个踩大坑系列！！！ 本来想在OS里面用汇编调用ecall来再进入系统调用来读取ramdisk的数据的，但是一方面汇编代码写的很多bug，导致一晚上没搞定，另外转念一想，我本来就在OS里面阿！！！我还调用个der呢，我直接来不是更方便更直接吗！！！然后问题就解决了～～
21. 在使用NDL库的时候遇到一个问题，那就是链接不上NDL库，即使我应用了NHL.h，通过makefile输出发现是链接的问题，那么就开始查找makefile文件，最后确认是make的时候没有添加libndl这个库。。。因为我是把dummy的文件夹拷贝过来的，这个细节卡了我好久。。。
22. 一切皆文件！！！在实现按键输入抽象的时候，就要把按键事件抽象成文件系统，通过NEMU的IOE，来实现按键事件的读取，输入到/dev/events这个抽象文件中（当然，这个文件只是虚构的，有这个名字而已，这样就可通过open打开获得fd，再调用它自身的read方法就可以实现事件的读取拉！！这就很像是面对对象的编程！），需要读取事件的时候就通过系统调用来读取，完成！至于为什么用open打开文件呢？因为open不带缓冲区，而fopen带缓冲，这样如果没有输入\n也能读取事件。
23. NDL读取屏幕相关的描述中，对于/proc/dispinfo这个文件的描述是否误导性太强了呢！！？？？看了半天，他的H和W不都还是从IOE获取的吗？我觉得有必要简化这里的描述
24. VGA接口问题，是不是太傻了。。。先是由坐标转换成offset到nanos中，然后调用IOE的时候，又要由offset转换回到坐标，其不是托裤子放屁？而且多了乘除法，效率太低。尽管有统一接口，但是太难受了。。。因此我选择一种特别的调用方式。
25. PA3.3
26. 这个浮点转定点的方式，为什么加0.5呢？？我这个确实搞不懂
27. 发现代码有一个地方有漏洞！那就是gpu显示这一块，代码中默认都是画布的大小和系统屏幕的大小是一致的。但是要是不一致就出问题了
28. 首先明确一个概念，当SDL\_Surface这个类被创建时，就已经把W和H设置好了。但是他设置的大小有可能和系统屏幕大小不一致！具体考虑是否直接loadbmp或者setvideomode，这个比较关键。如果直接loadbmp的话，他是会根据bmp大小尺寸来设置W和H的！所以最关键的就是，要在DrawRect之前，就要设置好画布的尺寸！！！即使用SDL\_SetVideoMode来实现！这样是最方便的
29. 突然发现，现在对我来说似乎printf和assert才是我找BUG的最大帮手～～。。。
30. 关于Flappy Bird里面load素材的load实现这个功能，其实可以自己使用stb\_image的库来实现。（框架代码应该就是这样做的，只不过做了封装而以）。
31. 搞PAL的时候经常遇到一些bug，发现有的很蠢，就是变量没有初始化。。。所以以后最好声明变量的时候要初始化，或者if语句记得要加else
32. wocao!!费尽很多心思，花了一天的事件来调试pal，终于解决了text BUG问题！！！原来是我蠢死了～～在SDL\_UpdateRect函数就有大bug了，原因就是在画一个矩形的时候，他取的数据也要根据canvas中对应的位置去取，而不是从头开始！！！至于为什么之前没有测试出来，那时因为flappy bird的每一次刷新图像都是整个canvas刷新的，所以刚好这个bug就没有影响，因为他是真的从0开始！！但是这次碰上了复杂的pal，就暴露出来了，并且一度让我抓狂崩溃！！！！
33. PA4
34. 内核切换还是挺难的呀！！一开始不理解框架代码搞得那么一堆数据结构，慢慢看慢慢画图理解终于弄懂了！！最重要的就是理解堆栈指针这个处理，以及上下文到底是如何保存的，怎么保存的！第一次切换的时候，是保存在OS自己的堆栈上的，之后才是保存到各自进程自己的堆栈上的！！！并且每一次都是通过SP指针的操作来确定保存位置，然后在schedule函数内保存到pcb的cp指针。切换的时候，这个CP指针被引用，因为他其实就是上次保存时候的sp，所以再通过这个恢复！一开始在kcontext函数内就要把各种值都配置好！
35. 用户进程更进一步，主要就是不知道怎么获得堆区的末地址。。。
36. 记住在程序加载之后，程序段0x83000000开始的地方会被新程序覆盖！！所以原来程序的数据就不可靠拉！！！
37. 操作系统真的太难学拉！！！！非常有必要专门为分页机制上运行多个用户进程写一个记录感想，因为知识点实在是太多辣！！
38. ye
39. 分页机制上运行用户程序，且能实现多个用户程序及内核程序切换！
40. 最难的就是可以运行多个用户程序了！！因为每个用户程序都有自己的栈、堆、分页表、上下文，因此要实现多用户程序切换，首先要在上下文切换实现好栈的切换、分页表的切换(satp寄存器)，最重要的是要解决malloc的问题！！！因为malloc是依靠sbrk系统调用实现的，而内存空间分配又是os的事情了，所以复杂就复杂在这个malloc函数！！坑了我好多天来调试。我的解决策略是通过map函数解决的，如果给map的pa参数传入的是NULL，那么就让map自己视情况来判断是否需要添加页，并且这种情况，不会造成资源浪费！（除了没有实现内存回收机制）这样我们实现了两个用户进程的并法执行