# 计算机系统设计

P32实验报告

专业：计算机科学与技术 学号：2013210 姓名：郑凌伟

1. 实验目的

实现一个足够简单的操作系统，来支持游戏“仙剑奇侠传”的运行

1. 实验内容

阶段1: 完成

1. 阶段一

首先我们按照要求实现loader，就是利用ramdisk提供的函数即可

1. #include "common.h"
3. #define DEFAULT\_ENTRY ((void \*)0x4000000)
5. **extern** **void** ramdisk\_read(**void** \*buf, off\_t offset, **size\_t** len);
7. **extern** **size\_t** get\_ramdisk\_size();
9. **uintptr\_t** loader(\_Protect \*as, **const** **char** \*filename) {
10. ramdisk\_read((**void**\*)0x4000000, 0, get\_ramdisk\_size());
11. **return** (**uintptr\_t**)DEFAULT\_ENTRY;
12. }

接下来就是实现中断机制了,首先需要实现lidt和int指令,将main.c中的宏定义解除注释,然后在decode中加入对int的支持并完成idtr结构体的构建.其中比较值得注意的是intr.c的实现,这就是我们对于中断的处理,具体来说,它的参数有两个：NO 是中断号，标识出要触发的特定中断或异常；ret\_addr 是返回地址，也就是中断处理程序完成后，程序应该继续执行的地址。然后它完成了如下步骤:  
1. 首先，保存当前的处理器状态。这包括标志寄存器（cpu.eflags），代码段寄存器（cpu.cs）和返回地址（ret\_addr）。这些都被压入栈，以便在中断处理程序完成后可以恢复。

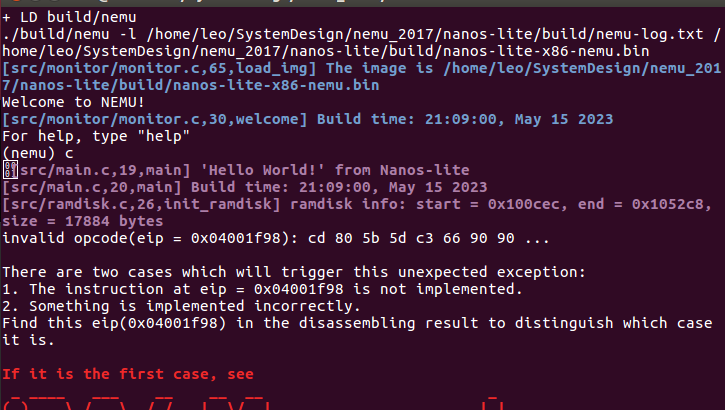
1. 然后，计算中断描述符表（IDT）中对应中断号的中断门的地址。中断描述符表是一个表格，存储着每个中断号对应的中断门的信息。中断门的信息分成两个部分，分别在 temp1 和 temp2 中。
2. 接着，从 temp1 和 temp2 中提取出中断处理程序的起始地址（jumptarget）。
3. 最后，设置 decoding.is\_jmp 为 1，表示要进行跳转，然后设置 decoding.jmp\_eip 为中断处理程序的起始地址。这样，下一步执行的就是中断处理程序了。
4. **void** raise\_intr(uint8\_t NO, vaddr\_t ret\_addr) {
5. /\* TODO: Trigger an interrupt/exception with ``NO''.
6. \* That is, use ``NO'' to index the IDT.
7. \*/
8. uint32\_t temp1;
9. uint32\_t temp2;
10. vaddr\_t jumptarget;
12. t0 = cpu.eflags;
13. rtl\_push(&t0);
14. t0 = cpu.cs;
15. rtl\_push(&t0);
16. t0 = ret\_addr;
17. rtl\_push(&t0);
19. temp1 = vaddr\_read(cpu.idtr.base + 8 \* NO, 4);
20. temp2 = vaddr\_read(cpu.idtr.base + 8 \* NO + 4, 4);
21. jumptarget = ((temp1 & 0x0000FFFF) | (temp2 & 0xFFFF0000));
23. decoding.is\_jmp = 1;
24. decoding.jmp\_eip = jumptarget;
25. }

之后我们需要实现pusha命令并重新租着TrapFrame结构体.

我们需要先填充iret指令的内容,然后完成内容:

1. rtl\_pop(&t0);：从栈中弹出一个值，存储到临时变量 t0 中。这个值是中断发生前的程序计数器值，也就是中断发生前处理器将要执行的下一个指令的地址。
2. decoding.jmp\_eip = t0; 和 decoding.is\_jmp = 1;：设置程序计数器为 t0，并标记要进行跳转。这样在 iret 指令执行完毕后，处理器将会继续执行 t0 指向的指令。
3. rtl\_pop(&t0); 和 cpu.cs = t0;：从栈中弹出一个值，存储到代码段寄存器中。代码段寄存器是用来存储当前执行的代码的内存区段的基址的。
4. rtl\_pop(&t0); 和 cpu.eflags = t0;：从栈中弹出一个值，存储到标志寄存器中。标志寄存器是用来存储处理器状态的寄存器，如进位标志、零标志、符号标志等。
5. make\_EHelper(iret) {
6. rtl\_pop(&t0);
7. decoding.jmp\_eip = t0;
8. decoding.is\_jmp = 1;
9. rtl\_pop(&t0);
10. cpu.cs = t0;
11. rtl\_pop(&t0);
12. cpu.eflags = t0;
13. print\_asm("iret");
14. }

这是当前阶段完成的效果



在第一阶段的最后我们需要实现系统调用,按照实验指导书的步骤,首先是在do\_event()中识别出系统调用事件\_EVENT\_SYSCALL, 然后调用do\_syscall().

1. #include "common.h"
3. \_RegSet\* do\_syscall(\_RegSet \*r);
5. **static** \_RegSet\* do\_event(\_Event e, \_RegSet\* r) {
6. **switch** (e.event) {
7. **case** \_EVENT\_SYSCALL: **return** do\_syscall(r);
8. **default**: panic("Unhandled event ID = %d", e.event);
9. }
11. **return** NULL;
12. }
14. **void** init\_irq(**void**) {
15. \_asye\_init(do\_event);
16. }

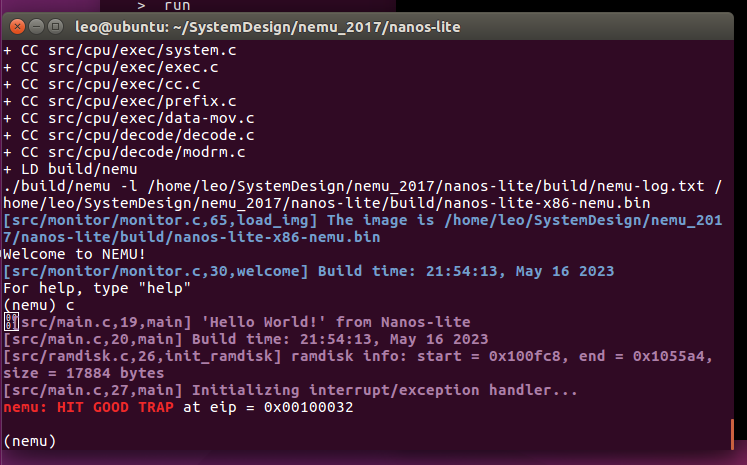
它接收一个事件e和寄存器集r作为参数。这个函数首先通过一个switch语句判断事件的类型。如果事件是一个系统调用事件（即\_EVENT\_SYSCALL），那么它就调用do\_syscall函数处理这个事件，并返回处理结果。如果事件的类型是其他未处理的类型，那么这个函数就会触发一个panic，表示发生了一个未处理的事件。

接着在arch中设置对应的宏,在修改前，SYSCALL\_ARGn(r)宏始终返回0，这意味着系统调用的参数无法正确获取。在修改后，这些宏被重新定义以返回对应的寄存器值（eax，ebx，ecx和edx），使得系统调用可以接收到正确的参数。

然后在main中解除ASYE的宏限制,另一个比较关键的点就是raise\_intr,具体实现如下功能

1. 首先，检查中断向量号（NO）是否超过中断描述符表（IDT）的限制。如果超过，则触发断言错误。
2. 之后，把cpu.eflags，cpu.cs以及ret\_addr压入堆栈。这些值保存了发生中断前的状态，以便于在中断处理完成后能恢复到中断发生前的状态。
3. 计算中断描述符在IDT中的地址，然后读取中断描述符的低32位和高32位。中断描述符的具体结构可以在i386手册中找到，这里简单地将低16位和高16位组合起来，形成了中断服务例程的地址。
4. 最后，设置跳转地址为中断服务例程的地址，并标记为需要跳转。

修改完成之后,我们再次运行,即可看到我们的程序正确运行并hit good trap了



1. 阶段二

在第二阶段,我们按照实验指导书首先要完成write的系统调用并运行helloworld小程序.

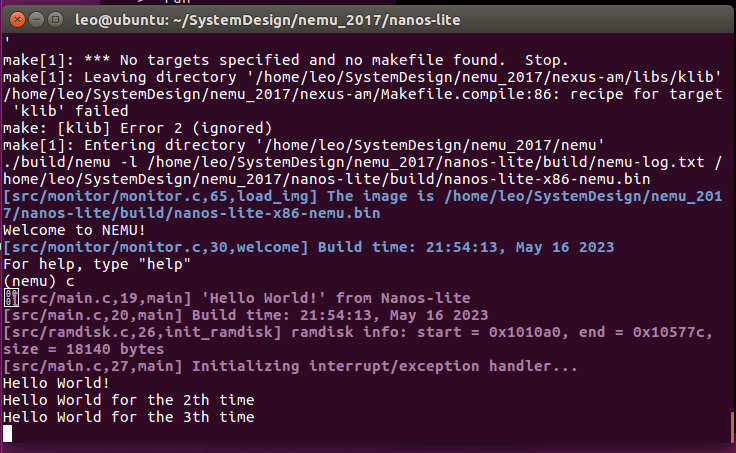
首先修改makefile指向helloworld,然后就是在syscall.c中实现sys\_write函数,关键主体如下所示:

1. **static** **inline** \_RegSet\* sys\_write(\_RegSet \*r) {
2. **int** fd = (**int**)SYSCALL\_ARG2(r);
3. **const** **char** \*buf = (**const** **char** \*)SYSCALL\_ARG3(r);
4. **size\_t** count = (**size\_t**)SYSCALL\_ARG4(r);
5. **if** (fd != 1 && fd != 2)
6. TODO();
7. **int** i;
8. **for** (i = 0; i < count; ++i)
9. \_putc(buf[i]);
10. SYSCALL\_ARG1(r) = i + 1;
11. **return** NULL;
12. }

在这个函数中，它首先从寄存器中获取参数，包括文件描述符（fd），写入的数据的地址（buf）和数据的长度（count）。然后，它检查fd是否为1或2，这两个值分别表示标准输出和标准错误。如果fd是其他值，那么函数暂时无法处理，因此调用TODO()。

接下来，函数使用一个循环将buf指向的数据逐个字符地写入到输出中，这是通过调用\_putc函数实现的。最后，它将写入的字符数加1赋值给SYSCALL\_ARG1(r)（即eax寄存器），这是write系统调用的返回值，表示写入的字符数。

至此sys\_write就实现好了.



紧接着我们需要实现堆区管理,具体来说在Nanos-lite中实现SYS\_brk系统调用, 然后在用户层实现\_sbrk().

我们先完成sys\_brk,在syscall中首先将SYSCALL\_ARG2(r)的值赋给了\_heap.end，这是改变数据段大小的操作。然后，它将0赋值给SYSCALL\_ARG1(r)，这是brk系统调用的返回值，表示操作成功

1. **static** **inline** \_RegSet\* sys\_brk(\_RegSet \*r) {
2. \_heap.end = (**void** \*)SYSCALL\_ARG2(r);
3. SYSCALL\_ARG1(r) = 0;
4. **return** NULL;
5. }

然后是sbrk的实现,首先，代码引入了一个全局变量 program\_break，该变量初始化为 end 的地址。end 是一个链接时定义的符号，表示程序数据段的末尾。

\_sbrk 函数通过 \_syscall\_ 函数发送 SYS\_brk 系统调用来改变程序的数据段大小。\_syscall\_ 的第二个参数为新的 program\_break 的地址。

如果 \_syscall\_ 返回值为0，表示系统调用成功，那么更新 program\_break，并返回原来的 program\_break。否则返回 (void \*)-1，表示分配内存失败。

1. **extern** **char** end;
2. **intptr\_t** program\_break = (**intptr\_t**)&end;
4. **void** \*\_sbrk(**intptr\_t** increment){
5. // char buf[100];
6. // sprintf(buf, "%x\n", &end);
7. // write(1, buf, strlen(buf));
8. **intptr\_t** old\_program\_break = program\_break;
9. **intptr\_t** addr = program\_break + increment;
11. **if**(\_syscall\_(SYS\_brk, addr, 0, 0) != 0)
12. **return** (**void** \*)-1;
13. program\_break = addr;
14. **return** (**void** \*)old\_program\_break;
15. }

接着就是实现完整的程序系统了,代码难度主要体现在fs.c,具体来说我们分门别类的实现了指导书中的若干fs函数:

Finfo 结构体被增加了一个 open\_offset 成员，用于记录文件的当前读写位置。

fs\_open 函数通过遍历 file\_table 来查找文件名对应的文件描述符（fd）。如果找到，就将文件的 open\_offset 重置为0，并返回该文件描述符。

fs\_read 函数通过文件描述符（fd）从文件的当前位置（由 open\_offset 决定）开始读取指定长度的数据。如果文件剩余的数据长度不足，那么只读取剩余的数据。读取数据后，open\_offset 要增加相应的长度。函数返回实际读取的数据长度。

fs\_write 函数和 fs\_read 类似，只是它是将数据写入文件。如果文件的剩余空间不足，那么只写入剩余的空间。写入数据后，open\_offset 也要增加相应的长度。函数返回实际写入的数据长度。

fs\_lseek 函数用于改变文件的当前读写位置。它有三种模式：SEEK\_SET 模式将读写位置设置为文件开头的偏移位置；SEEK\_CUR 模式将读写位置增加一个偏移量；SEEK\_END 模式将读写位置设置为文件末尾加上一个偏移量。

fs\_close 函数在这里并没有实现任何功能，只是返回0表示成功。

fs\_filesz 函数返回指定文件的大小。

实现了文件系统以后,就要让loader使用文件系统了,原始版本的加载器直接从 RAM 磁盘中读取整个可执行文件到默认入口地址 (DEFAULT\_ENTRY)。RAM 磁盘的读取通过 ramdisk\_read 实现，并且整个 RAM 磁盘的大小是通过 get\_ramdisk\_size 获取的。

修改后的版本改为从文件系统中读取文件。加载器首先打开文件，然后读取文件的全部内容到默认入口地址，最后关闭文件。这些操作分别通过 fs\_open，fs\_read 和 fs\_close 实现。文件的大小是通过 fs\_filesz 获取的。

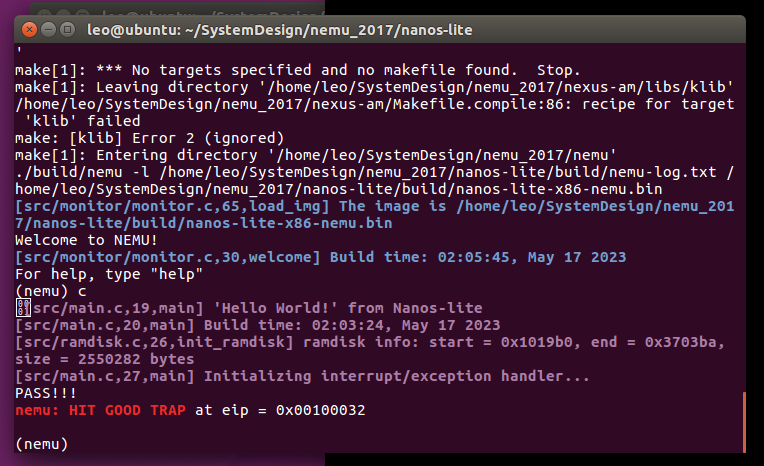
然后修改syscall,SYS\_open 调用会打开一个文件，返回一个文件描述符（fd）。这个调用需要三个参数：文件路径（pathname），打开方式（flags），和权限模式（mode）。

SYS\_read 调用会从一个已打开的文件（通过文件描述符指定）中读取数据。这个调用需要三个参数：文件描述符（fd），读取数据的缓冲区（buf），和读取的字节数（len）。

SYS\_close 调用会关闭一个已打开的文件。这个调用需要一个参数：文件描述符（fd）。

SYS\_lseek 调用会改变文件的当前读/写位置。这个调用需要三个参数：文件描述符（fd），偏移量（offset），和起始位置（whence）。起始位置可以是文件开始（SEEK\_SET），当前位置（SEEK\_CUR），或文件结束（SEEK\_END）。

最后可以看到我们的程序pass了样例



1. 阶段三

第三阶段是完善文件系统,首先是将VGA显存抽象成文件,我们从device开始入手.

dispinfo\_read函数用于读取显示设备的信息。修改后的版本使用memcpy函数，将dispinfo数组从offset开始的len字节的数据复制到buf中。

fb\_write函数用于将图形数据写入帧缓冲区（frame buffer），以在屏幕上显示。修改后的版本首先确认offset和len是4的倍数，然后计算出写入的起始位置（x和y），最后调用\_draw\_rect函数将图形数据写入帧缓冲区。

init\_device函数用于初始化设备。修改后的版本首先调用\_ioe\_init函数初始化输入输出设备，然后使用sprintf函数将屏幕的宽度和高度写入dispinfo数组。

接着我们来完善fs文件系统,完成了如下功能:init\_fs函数中，使用\_screen.width \* \_screen.height \* sizeof(uint32\_t)的方式来计算并初始化/dev/fb的大小。这个大小应该等于屏幕的像素数乘以每个像素的字节数。

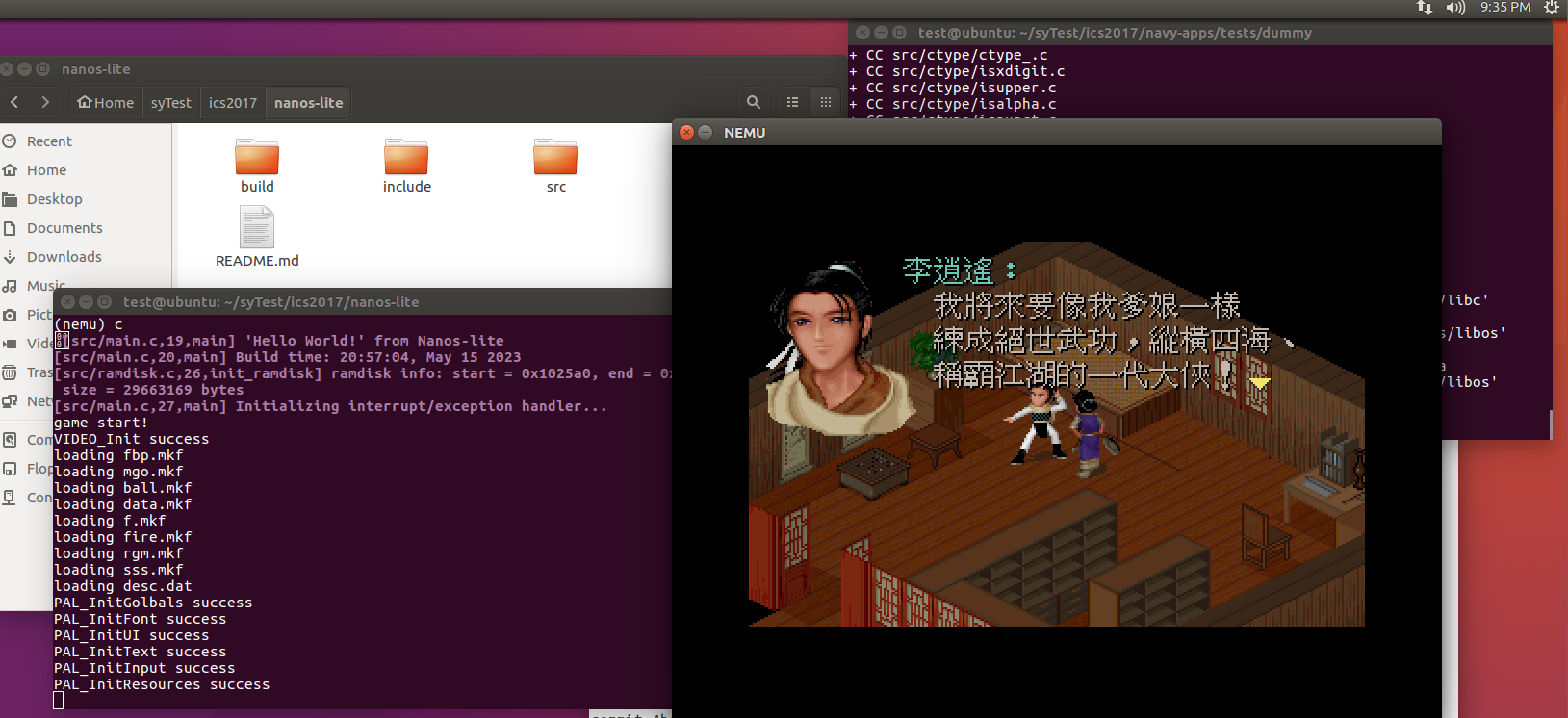
fs\_read函数中，如果文件描述符是FD\_DISPINFO，则调用dispinfo\_read函数读取显示设备信息，否则调用ramdisk\_read函数读取普通文件的内容。

fs\_write函数中，如果文件描述符是FD\_FB，则调用fb\_write函数将图形数据写入帧缓冲区，否则调用ramdisk\_write函数写入普通文件的内容。

之后我们还对一些细枝末节的代码进行了修改.

最后就是激动人心的运行了”仙剑奇侠传”:







1. 难点和总结

本次ICS2017 PA3实验主要涉及异常控制流和简易文件系统的实现，需要通过int 0x80中断机制将NEMU的实现，AM实现，操作系统的实现串联起来。其中，涉及的问题包括符号扩展、操作数长度，以及通过系统调用实现的打印函数等。此外，调试过程中可能会遇到较多的bug，需要仔细分析和解决。实验过程中，我加深了对于系统调用生命周期的理解，例如，通过navy-apps/tests/hello例子来理解从printf到系统调用\_write的过程。此外，还更加认识到理解NEMU和AM如何协同工作，以及在不同情况下printf的实现。

1. 必答题

文件读写的具体过程 仙剑奇侠传中有以下行为:

* 在navy-apps/apps/pal/src/global/global.c的PAL\_LoadGame()中通过fread()读取游戏存档
* 在navy-apps/apps/pal/src/hal/hal.c的redraw()中通过NDL\_DrawRect()更新屏幕

请结合代码解释仙剑奇侠传, 库函数, libos, Nanos-lite, AM, NEMU是如何相互协助, 来分别完成游戏存档的读取和屏幕的更新.

答:对于在仙剑奇侠传中涉及的文件读取和屏幕更新，这两个过程都涉及到多个部分的协同工作。在navy-apps/apps/pal/src/global/global.c的PAL\_LoadGame()函数中，数据的加载过程最终都依赖于fread()函数，这是一个在navy-apps/libs/libc/src/stdio/fread.c中实现的C语言库函数。在fread()函数中，通过调用memcpy()函数来读取文件内容，而文件的内容在调用fopen()函数时进行处理。

在navy-apps/apps/pal/src/hal/hal.c的redraw()函数中，利用NDL\_DrawRect()函数来刷新屏幕。NDL\_DrawRect()函数在navy-apps/libs/libndl/src/ndl.c中定义，它会调用fwrite()函数，这也是一个在navy-apps/libs/libc/src/stdio/fwrite.c中实现的C语言库函数。fwrite()函数通过一系列的函数调用，最终在navy-apps/libs/libos/src/nanos.c中的\_write()函数，由此将调用链路接到了我们实现的中断处理部分。此时，\_syscall()执行int 0x80参数中断，并设定好相应参数，随后操作系统通过nanos-lite/src/syscall.c中定义的\_RegSet\* do\_syscall(\_RegSet \*r)捕获中断，判断出其为SYS\_write,然后调用fs\_write()。在nanos-lite/src/fs.c中，我们通过参数fd判断出文件类型为FD\_fb，并通过fb\_write()更新屏幕。而fb\_write()则是调用了之前在nexus-am/am/arch/x86-nemu/src/ioe.c中实现\_draw\_rect()来在更新显示内容所对用的内存。

总结一下，应用程序的运行流程如下：在nanos-lite执行make run时，首先程序编译navy-apps路径下我们指定的应用程序，而在编译应用程序时使用的系统调用有关的实现均在navy-apps/libs/下代码中实现，这部分代码会通过在nexus-am/am/arch/x86-nemu/src/中定义的与硬件紧密相关的代码来实现对硬件的具体读写等操作，最终在应用程序、操作系统、lib、am等完成编译后生成镜像，在交给nemu执行。