PA3实验报告

231300027朱士杭

实验进度

已经可以在nanos-lite上面运行仙剑奇侠传，但是将am-kernels内容编译到navy中然后再nanoslite上面运行还没有实现，由于需要将am中所有ioe设备接口在libam上全部再实现一遍相当于重新做了一遍PA2的第三阶段，会拖慢整体进度，因而暂且搁置。然后已经在PA2里面实现声卡了，因为是选做内容，因而姑且搁置，等到了PA4的时候再回来填坑，然后Sys_execve系统调用那一块正在写，但是好像出了一点点问题，为了不搁置进度，因此先把实验报告完成了，然后顺便在写实验报告的过程中重新审视一下源代码找找问题出在哪里。

更新一下当前新的实验进度，已经实现了libam库函数接口了，可以在nanoslite上面运行am-kernels的程序了，然后也已经完成了Sys_execve以及上下层的系统执行函数，可以实现从Nterm/Nslider开始与结束并调用其他程序。

必答题

理解上下文结构体的前世今生

观察trap.S文件，我们一条一条详细解释（这个文件通常是由操作系统的内核开发者编写的，用于处理来自硬件或软件的中断请求。它确保在中断发生时，当前的程序状态被正确保存，并在中断处理完成后恢复程序状态）

```
__am_asm_trap:
  addi sp, sp, -CONTEXT_SIZE #将栈指针（sp）减去CONTEXT_SIZE，为保存上下文腾出空间。CONTEXT_SIZE是保存所有寄存器所需的栈空间大小

  MAP(REGS, PUSH) #这个宏用于将寄存器的当前值保存到栈上。PUSH操作通常意味着将数据压入栈中，用于将寄存器值推送到栈上。

  csrr t0, mcause #读取mcause寄存器的值到临时寄存器t0。mcause寄存器包含异常或中断的原因
  csrr t1, mstatus #读取mstatus寄存器的值到临时寄存器t1。mstatus寄存器包含机器状态寄存器的值，其中包括当前的特权级别和一些控制位。
  csrr t2, mepc #读取mepc寄存器的值到临时寄存器t2。mepc寄存器包含异常发生时的程序计数器（PC）值。

  STORE t0, OFFSET_CAUSE(sp) #将t0寄存器（包含mcause的值）存储到栈上，偏移量为OFFSET_CAUSE。
  STORE t1, OFFSET_STATUS(sp) #将t1寄存器（包含mstatus的值）存储到栈上，偏移量为OFFSET_STATUS。
  STORE t2, OFFSET_EPC(sp) #将t2寄存器（包含mepc的值）存储到栈上，偏移量为OFFSET_EPC。

  # set mstatus.MPRV to pass difftest
  li a0, (1 << 17) #将值1 << 17加载到寄存器a0。这个值用于设置mstatus寄存器中的MPrv位
  or t1, t1, a0 #将t1寄存器（包含mstatus的值）与a0寄存器的值进行逻辑或操作，并将结果存回t1。这用于设置mstatus寄存器中的MPrv位。
  csrw mstatus, t1 #将t1寄存器的值写回mstatus寄存器。

  mv a0, sp  #将栈指针sp的值移动到寄存器a0。这通常是为了将栈指针传递给异常处理函数。
```

```
    call __am_irq_handle #调用异常处理函数__am_irq_handle

    LOAD t1, OFFSET_STATUS(sp) #从栈上偏移量为OFFSET_STATUS的位置加载值到t1寄存器。
    LOAD t2, OFFSET_EPC(sp) #从栈上偏移量为OFFSET_EPC的位置加载值到t2寄存器。
    csrw mstatus, t1  #将t1寄存器的值写回mstatus寄存器，恢复机器状态。
    csrw mepc, t2 #将t2寄存器的值写回mepc寄存器，恢复程序计数器。

    MAP(REGS, POP) #这个宏用于从栈上恢复寄存器的值。POP操作通常意味着从栈中弹出数据。

    addi sp, sp, CONTEXT_SIZE #将栈指针sp增加CONTEXT_SIZE，恢复栈空间。
    mret #从异常或中断返回，继续执行被中断的程序。
```

此时执行完将t0,t1,t2寄存器里面的值保存到栈上之后就可以调用__am_irq_handle函数了，而那个Context* c其实指向的就是对应的栈上的值，调用完之后再继续恢复机器状态（包括寄存器的值）继续执行。

因此在这里就是需要涉及到Context结构体内部的顺序问题了，根据压栈的顺序不同，按照t0(mcause),t1(mstatus),t2(mepc)的顺序进行压栈，而且在trap.S中还有如下宏定义:

```
#define CONTEXT_SIZE  ((NR_REGS + 3) * XLEN)
#define OFFSET_SP     ( 2 * XLEN)
#define OFFSET_CAUSE  ((NR_REGS + 0) * XLEN)
#define OFFSET_STATUS ((NR_REGS + 1) * XLEN)
#define OFFSET_EPC    ((NR_REGS + 2) * XLEN)
```

表明相对于sp栈顶指针的偏移量是多少，从而可以计算出顺序应当是mcause、mstatus、mepc

另外，不要忘了Context结构体里面还有`gpr[NR_REGS]`这个寄存器组的上下文内容，那么这个是在哪里赋值的呢？不要忘了还有一个`MAP(REGS, PUSH)`这个函数，就是用来将寄存器的当前值保存到栈上

诶话说我的Context结构体定义里面怎么还有一个`void *pdir;`啊，这个是什么来着，我突然忘了……没事这里先挖个坑，待会可能需要回来补……

理解穿越时空的旅程

因为从命令行键入mainargs=i输出的过程已经在PA2里面写过了，因此这里就不再赘述调用IOE的过程了，直接从yield test里面void hello_intr() 函数调用yield()函数开始讲吧（我自己对代码框架进行了一些调整先空转然后再进行yield操作，否则好像会出一些问题）

开始执行yield()函数，首先先明确是调用了`abstract-machine/am/src/riscv/nemu/cte.c`当中的yield()函数，而这里yield函数内部其实只有一条asm手动插入的汇编代码`asm volatile("li a7, -1; ecall");`那这条代码的意思就是执行ecall指令，将-1加载进a7寄存器，那么-1是什么呢？-1其实就是mcause也就是异常发生号，从而我们就能得知，a7寄存器就是用来存放异常号的。

然后nemu就会执行这条`li a7, -1; ecall`指令，查看inst.c文件中有关于ecall的指令实现`INSTPAT("0000000 00000 00000 000 00000 11100 11", ecall ,N, ECALL;);`然后就会调用ECALL宏命令`#define ECALL s->dnpc = (isa_raise_intr(gpr(17),s->pc))`然后就会调用isa_raise_intr()函数

然后isa_raise_intr()函数的实现又是在`nemu/src/isa/riscv32/system/intr.c`文件当中， 具体实现过程如下面代码所示：

```
word_t isa_raise_intr(word_t NO, vaddr_t epc) {  //触发了一个异常/中断 asm
  /* TODO: Trigger an interrupt/exception with ``NO''.    触发一个带有“NO”的中断/异常* Then return the address of the interrupt/exception vector.  然后返回中断/异常向量的地址*/
  //如果会抛出一个异常/中断的话，那调用指令的时候就会调用ecall指令，而且调用ecall指令是一定会调用isa_raise_intr函数的，那么估计etrace功能就应该是放在这里的
  #ifdef CONFIG_ETRACE
  etrace_errors(NO, epc);//这里进行etrace的基础设施实现
  #endif
  switch (NO)
  {case -1: //表示这个时候是需要加4的  因为是yield自陷  看asm手动插入的那一条汇编语言代码
  case 0:case 1:case 2:case 3:case 4:case 5:case 6:case 7:case 8:case 9:case 10:
  case 11:case 12:case 13:case 14:case 15:case 16:case 17:case 18:case 19:
//先暂时默认所有的都需要+4    要是之后有需要调整的再修改……这个一开始只有case0和1结果又报错了……唉……无语死了
  epc+=4; break;   //当case为0和1的时候同样需要epc+=4否则又重新陷入ecall了……
  default:   break;}
  cpu.CSRs.mcause=NO;//存储触发异常的原因NO
  cpu.CSRs.mepc=epc; //存储触发异常的pc(如果经历了+4的话就是存储mret的时候下一条将要执行的指令)
  return cpu.CSRs.mtvec;//从mtvec寄存器中取出异常入口地址并返回
}
```

在isa_raise_intr函数当中需要改变epc的值，也就是传入的发生异常的pc的值，将其改变为mret的时候下一条将要执行的指令地址，另外还需要存储触发异常的原因NO，并且从mtvec寄存器中取出异常入口地址并返回给s->dnpc也就是接下来下一步要执行的指令地址

注意，在一开始当我们选择`yield test`时, `am-tests`会通过`cte_init()`函数对CTE进行初始化, 其中包含一些简单的宏展开代码。这最终会调用位于`abstract-machine/am/src/$ISA/nemu/cte.c`中的`cte_init()`函数。 `cte_init()`函数会做两件事情, 第一件就是设置异常入口地址:对于riscv32来说, 直接将异常入口地址设置到mtvec寄存器中即可。 `cte_init()`函数做的第二件事是注册一个事件处理回调函数, 这个回调函数由`yield test`提供。

那么具体的，这个`yield test`提供的回调函数是什么呢？观察`main.c`里面的`CASE('i', hello_intr, IOE, CTE(simple_trap));`可以发现`simple_trap`就是这个提供的回调函数

```
Context *simple_trap(Event ev, Context *ctx) {
  switch(ev.event) {
    case EVENT_IRQ_TIMER:
      putch('t'); break;
    case EVENT_IRQ_IODEV:
      putch('d'); break;
    case EVENT_YIELD:
      putch('y'); break;
    // case EVENT_ERROR:
    //   putch('e'); break;
    default:
      printf("未处理的event为:%d\n",ev.event);panic("Unhandled event\n");
break;
```

```
  }
  return ctx;
}
```

让我们再回到isa_raise_intr()函数返回之后吧，将mtvec寄存器中取出异常入口地址并返回给s->dnpc之后，那么下一条执行的指令就是mtvec寄存器中的值。那么问题又来了，mtvec寄存器中的值是什么呢？我找了半天好像也没有找到设置mtvec寄存器值的相关代码……又出现坑了……等之后再回来填坑吧。暂且假设mtvec寄存器值就是trap.S当中__am_asm_trap过程的地址好了

这里插入一条，后来想明白了这个mtvec寄存器的值是怎么来的了，在`abstract-machine/am/src/riscv/nemu/cte.c`文件当中的`bool cte_init(Context*(*handler)(Event, Context*))`函数当中有一条初始化初始化异常入口的汇编指令，设置mtvec寄存器值为__am_asm_trap过程的地址：

```
asm volatile("csrw mtvec, %0" : : "r"(__am_asm_trap));
```

然后就开始执行__am_asm_trap过程，经过一些前期存储上下文的操作（在理解上下文结构体的前世今生中已经解释过了这里不再赘述），然后就是执行`Context* __am_irq_handle(Context *c)`函数了

```
Context* __am_irq_handle(Context *c) {
  if (user_handler) {
    Event ev = {0};
    switch (c->mcause) {//根据c->mcause上下文信息判断对应的事件是什么  （即事件分发）
      case -1:  ev.event= EVENT_YIELD; break;
      //观察navy-apps/libs/libos/src/syscall.h可得  当c->mcause为1~19时都属于
syscall
      case 0:case 1:case 2:case 3:case 4:case 5:case 6:case 7:case 8:case
9:case 10:
      case 11:case 12:case 13:case 14:case 15:case 16:case 17:case 18:case
19:
      ev.event=EVENT_SYSCALL;break;
      default:  ev.event = EVENT_ERROR; break;//正是因为自己没有识别出自陷异常的操
作，因此才会报错
    }
    //然后执行user_handler函数   即cte_init中传入的handler函数
    c = user_handler(ev, c);
    assert(c != NULL);
  }
  return c;
}
```

根据c->mcause的异常号，在这里执行yield也就是-1，给事件分配为`ev.event= EVENT_YIELD;`然后执行`c = user_handler(ev, c);`函数，注意这里的user_handler函数也就是yield test提供的处理异常的回调函数`simple_trap`，这个函数在`am-kernels/tests/am-tests/src/tests/intr.c`中提供：

```
Context *simple_trap(Event ev, Context *ctx) {
  switch(ev.event) {
```

```
      case EVENT_IRQ_TIMER:
        putch('t'); break;
      case EVENT_IRQ_IODEV:
        putch('d'); break;
      case EVENT_YIELD:
        putch('y'); break;
      default:
        printf("未处理的event为:%d\n",ev.event);panic("Unhandled event\n");
  break;
    }
    return ctx;
  }
```

根据`ev.event= EVENT_YIELD;`可得switch语句应当执行`putch('y');`功能输出一个y，然后把ctx原模原样返回（之后可能会进行一些特殊处理，但是这里目前先不动）。从simple_trap函数返回之后，然后又从__am_irq_handle函数返回，这样再来看trap.S中的内容，恢复机器执行_am_irq_handle函数之前的行为，然后再mret进行返回。

那么在nemu当中的mret指令执行如下：

```
INSTPAT("0011000 00010 00000 000 00000 11100 11", mret  ,N, {s-
>dnpc=cpu.CSRs.mepc;});
```

下一条指令就是cpu的CSRs控制状态寄存器中的mepc寄存器的值，不要忘了，这个值可是被存到栈上去过然后又被存回t2寄存器了，而这个寄存器原本存的就是改变过后的发生异常的pc的值（在这里就是发生异常的pc地址+4的值），这样就可以实现处理完异常/中断之后继续执行下面的操作了。对于这段“穿越时空的旅程”，我只能说一句——绝，真的太绝了。

hello程序是什么, 它从而何来, 要到哪里去

（由于写实验报告的时候已经完成了文件系统了，已经不太记得当时是怎么做的了，因而有些回答可能会有一些问题）

hello程序一开始在哪里？`navy-apps/tests/hello/hello.c`是一个C源文件，根据Makefile文件与libc等库文件进行链接，然后在build文件夹下面产生了`navy-apps/tests/hello/build/hello-riscv32`可执行文件ELF格式，然后我们将其copy到`nanos-lite/build`文件夹下面并且改名为ramdisk.img文件，注意这个时候ramdisk.img虚拟存储映像里面只有hello一个程序，之后具体实现文件系统的时候会加入更多的文件进来

hello程序是怎么出现在内存中的以及为什么会出现在目前得内存位置？我们需要永远记住Nanos-lite本质上真实一个C语言程序运行在am之上的，观察`nanos-lite/src/main.c`函数当中，前面都是一些初始化init虚拟存储mm、设备device、磁盘ramdisk、中断处理请求irq、文件系统fs以及进程proc。这里最重要的函数就是初始化进程函`init_proc();`观察函数内部，又调用了`naive_uload(NULL,"/bin/hello");`函数，这个函数又在`nanos-lite/src/proc.c`当中，里面又调用了`uintptr_t entry = loader(pcb, filename);`函数，调用loader函数返回了entry入口地址。至于loader函数是如何具体执行的，主要就是装载ELF文件当中的段，这里就是属于“加载第一个应用程序”中的任务。

对于`static uintptr_t loader(PCB *pcb, const char *filename)` 函数，首先先`int fd=fs_open(filename,0,0);`打开`bin/hello`文件，然后读取ramdisk.img可执行文件ELF头部信息

```
//Ehdr 是 ELF Header 的缩写，它代表 ELF 文件的头部信息。ELF Header 包含了描述 ELF 文件如何被分析的信息，例如文件类型、机器类型、版本、入口点地址、程序头表和节头表的偏移量等重要信息。
  Elf_Ehdr elf_header;
  Elf_Ehdr* elf=&elf_header;
  // ramdisk_read(elf, 0, sizeof(Elf_Ehdr));//读取elf的header
  int read_elf_len=fs_read(fd,elf,sizeof(elf_header));
```

然后使用一个for循环，遍历每一个程序头然后计算出程序头的起始位置（基址），根据base基址去更新文件指针的位置，然后判断需要装载的段读取内存大小为p_memsz的程序头内容给对应地址为p_vaddr的段里面（当然同时需要注意将.bss节内容初始化为0）

```
if (ProgramHeaders[i].p_type == PT_LOAD) {//如果第i个程序头的种类是Load的话
  ramdisk_read((void*)ProgramHeaders[i].p_vaddr, ProgramHeaders[i].p_offset, ProgramHeaders[i].p_memsz);//读取内存大小为p_memsz的程序头内容给对应地址为p_vaddr的段里面
  memset((void*)(ProgramHeaders[i].p_vaddr+ProgramHeaders[i].p_filesz), 0, ProgramHeaders[i].p_memsz - ProgramHeaders[i].p_filesz);      // set .bss with zeros将.bss的section初始化为0
    // }
```

然后就是关闭文件`return elf->e_entry;`返回ELF文件中的entry入口地址

loader函数返回entry入口地址之后，执行`((void(*)())entry) ();`跳转到程序的入口地址，开始正式执行hello程序。接下来的内容其实与nanos-lite没什么关系了，主要内容都是hello程序内容了。而注意在Navy当中的Makefile当中有`LIBS += libc libos`将libc和libos库链接进来，之后hello程序当中的所有函数都是调用这些库函数里面的内容了，比如`printf("Hello World from Navy-apps for the %dth time!\n", i ++);`就是调用include进`navy-apps/libs/libc/src/stdio/printf.c`的printf.c文件内容，而“每一个字符又是经历了什么才会最终出现在终端上”这个内容又是涉及到了“C库中printf()到write()转换的部分”的部分了，这一部分就扣除吧不属于PA的主线内容。

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, 动画里仙鹤在群山中飞过. 这一动画是通过navy-apps/apps/pal/repo/src/main.c中的PAL_SplashScreen()函数播放的.

在PAL_SplashScreen()函数当中先是定义了一系列的变量

```
   SDL_Color     *palette = PAL_GetPalette(1, FALSE);//存储调色板的指针
   SDL_Color      rgCurrentPalette[256];             //当前调色板的数组。
   SDL_Surface   *lpBitmapDown, *lpBitmapUp;         //用于存储上部和下部图像的表面
   SDL_Rect       srcrect, dstrect;                  //用于定义源和目标矩形区域
   LPSPRITE       lpSpriteCrane;//存储鹤动画的精灵
   LPBITMAPRLE    lpBitmapTitle;// 存储标题图像的指针
   LPBYTE         buf, buf2;//用于存储解压缩后的图像数据的缓冲区
   int            cranepos[9][3], i, iImgPos = 200, iCraneFrame = 0, iTitleHeight;  // 存储鹤位置和动画帧的数组
```

```
    DWORD           dwTime, dwBeginTime;//用于跟踪时间的变量
    BOOL            fUseCD = TRUE;//用于指示是否使用 CD 音频的标志
```

然后获取调色板，如果获取失败，输出错误信息并返回

为了简化，一次性分配所有所需的内存，分配内存以存储图像数据。buf 用于存储图像数据，buf2 用于存储解压缩后的数据，lpSpriteCrane 用于存储鹤精灵。

```
    buf = (LPBYTE)UTIL_calloc(1, 320 * 200 * 2);
    buf2 = (LPBYTE)(buf + 320 * 200);
    lpSpriteCrane = (LPSPRITE)buf2 + 32000;
```

然后创建与屏幕兼容的表面surface，以便绘制图像。

```
    lpBitmapDown = VIDEO_CreateCompatibleSurface(gpScreen);
    lpBitmapUp = VIDEO_CreateCompatibleSurface(gpScreen);
```

接下来就是最主要的Read the bitmaps读取和解压缩位图，读取上部和下部的位图数据，解压缩并将其绘制到表面上，`PAL_MKFReadChunk`函数用于从 MKF 归档文件中读取指定块（chunk）的数据。从 MKF 归档文件中读取指定编号的块到提供的缓冲区，通过验证参数、获取块的偏移量和长度，以及执行实际的文件读取操作来实现这一功能。该函数处理了多种错误情况，并返回适当的错误代码。 `PAL_FBPBlitToSurface` 函数负责将压缩的位图数据直接复制到 SDL 表面。这个函数没有实际执行 RLE 解码，因为它的注释中提到了“RLE-compressed bitmap”，但实际上函数实现是直接复制像素数据，这意味着位图数据已经是解压的。函数通过简单的指针操作和循环来完成像素数据的复制，确保位图正确地绘制到 SDL 表面上。

最主要的还是下面的代码，其中`gpGlobals->f.fpMGO`来自于`navy-apps/apps/pal/repo/data/mgo.mkf`

```
    PAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_TITLE, gpGlobals->f.fpMGO);
    PAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_CRANE, gpGlobals->f.fpMGO);
    Decompress(buf, lpSpriteCrane, 32000);//将buf解压缩到lpSpriteCrane鹤精灵
```

应用程序通过调用库函数（如 PAL_MKFReadChunk）来请求从 mgo.mkf 文件中读取特定的像素数据块。这个库函数封装了对文件操作的需求，使得应用程序不需要直接处理文件系统的复杂性

然后就是使用一个for循环随机生成9个鹤的位置和动画帧

```
    for (i = 0; i < 9; i++)
    {
      cranepos[i][0] = RandomLong(300, 600);
      cranepos[i][1] = RandomLong(0, 80);
      cranepos[i][2] = RandomLong(0, 8);
    }
```

在一个`while (TRUE)`的大循环当中，在前15秒内，逐渐调整调色板的颜色，用于实现渐变的效果

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

然后绘制图像，绘制上部和下部的位图，最重要的是绘制鹤的图像，至于标题就不赘述了

```
    for (i = 0; i < 9; i++)
    {
      LPCBITMAPRLE lpFrame = PAL_SpriteGetFrame(lpSpriteCrane,
        cranepos[i][2] = (cranepos[i][2] + (iCraneFrame & 1)) % 8);
      cranepos[i][1] += ((iImgPos > 1) && (iImgPos & 1)) ? 1 : 0;
      PAL_RLEBlitToSurface(lpFrame, gpScreen,
        PAL_XY(cranepos[i][0], cranepos[i][1]));
      cranepos[i][0]--;
    }
    iCraneFrame++;
```

然后就是检查用户是否按下了某个键，如果按下，则退出循环；如果不是的话就进入下一个循环绘制下一帧的图像。

由于本问题当中我们主要关注的是各个抽象层之间如何进行相互协助帮助PAL从mgo.mkf文件中读出仙鹤的像素信息并且更新到屏幕上，我们主要关心的代码只有如下：

```
    PAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_TITLE, gpGlobals->f.fpMGO);
    PAL_MKFReadChunk(buf, 32000, SPRITENUM_SPLASH_CRANE, gpGlobals->f.fpMGO);
    Decompress(buf, lpSpriteCrane, 32000);
    for (i = 0; i < 9; i++){cranepos[i][0] = RandomLong(300, 600);cranepos[i]
[1] = RandomLong(0, 80);cranepos[i][2] = RandomLong(0, 8);}
    while(TRUE){
      for (i = 0; i < 9; i++){
        LPCBITMAPRLE lpFrame = PAL_SpriteGetFrame(lpSpriteCrane,cranepos[i][2]
= (cranepos[i][2] + (iCraneFrame & 1)) % 8);
        cranepos[i][1] += ((iImgPos > 1) && (iImgPos & 1)) ? 1 : 0;
        PAL_RLEBlitToSurface(lpFrame, gpScreen,PAL_XY(cranepos[i][0],
cranepos[i][1]));
        cranepos[i][0]--;}
      iCraneFrame++;}
```

现在我们来详细解释一下这段代码，在PAL_MKFReadChunk函数当中，先是验证 lpBuffer 和 fp 是否为 NULL，以及 uiBufferSize 是否为 0。如果任何检查失败，返回 -1。然后调用PAL_MKFGetChunkCount 函数获取 MKF 文件中的总块数。然后使用fseek函数（这样就可以使用外部stdio.c当中的库函数与外部IOE接口交互）将文件指针移动

到包含第 uiChunkNum 块偏移量的文件位置（4 * uiChunkNum），读取当前块的偏移量 uiOffset 和下一个块的偏移量 uiNextOffset，使用 SDL_SwapLE32 函数将读取的偏移量从 LE（Little Endian）小端模式转换为主机字节序。最后从文件中读取 uiChunkLen 长度的数据到 lpBuffer，返回读取的字节数，这些都与外部进行交互的函数比如fseek函数就需要在Navy里面进行系统调用_syscall_（在navy-apps/libs/libos/src/syscall.c当中使用到了return _syscall_(SYS_lseek, fd, offset, whence);的_lseek函数），然后nanoslite就捕获到了一次系统调用，在nanos-lite/src/syscall.c当中执行case SYS_lseek:c->GPRx = system_lseek(a[1], a[2] , a[3]);代码，从而转到size_t system_lseek(int fd, size_t offset, int whence){return fs_lseek(fd,offset,whence);}函数进行执行。这些本质上都是运行在am之上的，将其转化为ELF可执行文件之后再传递给NEMU去执行，只要涉及到与IOE与fs相关的内容都是这样子运行的，需要库函数, libos, Nanos-lite, AM这些相互协作，因为nanoslite与nemu是解耦的，因而可以暂时不用管nemu是如何运行的。

然后在大的while循环里面，有一个for循环，每一次循环都是先获取鹤的帧给lpFrame，然后按照8-bit的方式将帧画在Surface即画布上面，这里就会使用到libminiSDL中的video.c文件有关于画图的相关SDL_BlitSurface，SDL_UpdateRect函数，需要相关库函数的支持。而调用这些库函数的内容又需要进行系统调用，应用程序从Navy的libos中的syscall.c的_write函数出发触发系统调用，然后操作系统不够nanoslite做出反应在nanos-lite/src/syscall.c中执行case SYS_read:c->GPRx = system_read(a[1], a[2] , a[3]);进行试图进行与外部设备交互进行写的操作，因为有了fs之后所有的写操作归根到底最后都是调用了ramdisk_read函数与磁盘IOE设备打交道。然而这些都是运行在am之上的，在am的角度来看，就是在进行IOE相关操作，甚至于就是在与磁盘这个外部设备进行交互。am将其转换为ELF可执行文件之后就传给NEMU进行运行。这样子就可以实现从应用程序->操作系统->abstract machine->NEMU的传递相互协作了，从而帮助PAL从mgo.mkf文件中独处仙鹤的像素信息，并且更新到屏幕上。

思考题

AM究竟给程序提供了多大的栈空间

在提供的配置文件中，并没有直接指定栈空间大小的配置项。通常，栈空间的大小是在编译时通过编译器的选项来设置的，例如在GCC中可以使用-Wl,--stack,size来指定栈的大小。

然而，有一个配置项可能与内存大小有关，这可能间接影响到栈空间的大小，那就是：

```
#define CONFIG_MSIZE 0x8000000
```

这里CONFIG_MSIZE定义了整个虚拟内存的大小，其值为0x8000000，即128MB。但这并不是栈空间的大小，而是整个虚拟内存的大小。栈空间通常是从这个虚拟内存中分配的，但具体的大小和位置是由操作系统或程序的运行时环境决定的。

如果需要设置栈空间的大小，可能需要查看程序的源代码或者编译器的文档来找到正确的设置方法。在某些情况下，栈空间的大小是默认的，由操作系统或编译器自动管理。如果正在编写一个操作系统或者需要精确控制栈空间大小的程序，可能需要手动设置这个值。

那么在编译的过程中就需要用到abstract-machine/scripts/linker.ld这个链接文件了， 观察$AM_HOME/scripts/linker.ld这个链接脚本可以发现，其中定义了一个符号_stack_pointer

```
  _stack_top = ALIGN(0x1000);
  . = _stack_top + 0x8000;
  _stack_pointer = .;
```

```
    end = .;
    _end = .;
    _heap_start = ALIGN(0x1000);
```

这里有提及栈顶是往高处多0x1000空间开始是栈顶（估计是怕栈溢出之后影响了下面的程序吧）然后栈底比栈顶高0x8000空间，因此栈的大小就是0x8000字节

而根据AM启动客户程序的流程可知，在am/src/riscv/nemu/start.S中的_start:中将会执行la sp, _stack_pointer，以此初始化栈指针。又注意到_stack_top符号的地址与之相差0x8000，因此可知AM中程序的栈空间大小为0x8000字节

堆和栈在哪里

我们提到了代码和数据都在可执行文件里面, 但却没有提到堆(heap)和栈(stack).

那为什么堆和栈的内容没有放入可执行文件里面?因为堆和栈的内容是在运行的过程中动态产生的，是属于“进程”的概念，而代码和数据都是属于“程序”的概念是静态的。这就好比非static局部变量是在运行的过程中分配堆栈空间的，但是这个并不耗费可执行文件的大小，这些堆栈内容都是属于运行时环境支持，不用放入可执行文件里面

那么那程序运行时刻用到的堆和栈又是怎么来的?不要忘了，Nanos-lite本质上只是一个C语言程序，它所用到的堆栈其实是运行时环境支持的也就是am当中的trm.c的堆栈空间，这些资源都是am给予的而与nanos-lite无关。

如何识别不同格式的可执行文件 （GNU/Linux是如何知道"格式错误"的?）

可执行文件本质上也是一个ELF文件，而ELF文件都是有一个魔数（Magic Number）的，GNU/Linux在将可执行文件加载到主存当中时，会先检查魔数是否合法，因为不同操作系统的可执行文件的格式是不一样的，开头的四个字节所对应的魔数也必然是各不相同的，通过检查魔数就可以得知是否是GNU/Linux所允许的可执行文件格式。

冗余的属性? 使用readelf查看一个ELF文件的信息, 你会看到一个segment包含两个大小的属性, 分别是FileSiz和MemSiz, 这是为什么? 再仔细观察一下, 你会发现FileSiz通常不会大于相应的MemSiz, 这又是为什么?

在 ELF（Executable and Linkable Format）文件中，segment 指的是程序的一段区域，比如代码段（text segment）、数据段（data segment）、BSS 段等。readelf 工具可以用来查看 ELF 文件的多种信息，包括这些段的属性。

每个 segment 都有两个大小属性：

- FileSiz：这个属性表示 segment 在文件中的实际大小，即在磁盘上占用的空间。它包括了 segment 中所有非零的数据。如果 segment 中有一部分是初始化为零的，这部分通常不会被写入文件中，以节省空间。
- MemSiz：这个属性表示 segment 在内存中的大小，即程序加载到内存时占用的空间。MemSiz 包括了 segment 中所有的空间，无论这部分是否被初始化为零。对于初始化为零的部分（如 BSS 段），虽然它们在文件中不占用空间（因为它们没有被写入文件），但在内存中仍然需要分配空间。

这就是为什么 FileSiz 通常不会大于 MemSiz 的原因：

- FileSiz 仅包括实际写入文件的非零初始化数据。
- MemSiz 包括了所有的空间，包括未初始化的部分（它们在文件中可能不占用空间）。

例如，如果一个 segment 包含了 1KB 的非零初始化数据和 3KB 的未初始化数据，那么 FileSiz 将是 1KB，而 MemSiz 将是 4KB。当程序加载到内存时，操作系统会为整个 segment 分配 4KB 的空间，包括那 3KB 的未初始化数据。

这种设计允许 ELF 文件在磁盘上占用更少的空间，同时确保程序在内存中有足够的空间运行。操作系统在加载程序时会根据 MemSiz 为每个 segment 分配足够的内存空间，并根据需要初始化这些空间。

为什么要清零? 为什么需要将 [VirtAddr + FileSiz, VirtAddr + MemSiz) 对应的物理区间清零?

将在.bss节的未初始化的全局变量初始化为0

检查ELF文件的魔数

在GNU/Linux当中ELF文件的魔数应当为`0x 7f 45 4c 46`即对应ASCII码的`删除Delete字符 E L F`， 实际上由于小端模式如果直接去判断前四个字节应当与 `0x464c457f` 作比较！！！！

检测ELF文件的ISA类型

要检查 ELF 文件的 ISA（指令集架构）类型，可以查看 ELF 文件头中的 e_machine 字段。e_machine 字段包含了一个代码，指明了 ELF 文件是为哪种处理器或ISA编译的。这个字段的值可以在 Elf64_Ehdr 结构体中找到，其类型为 Elf64_Half

例如，如果你使用 readelf 工具，可以通过以下命令查看 ELF 文件头信息，其中包含了 e_machine 字段：

```
readelf -h yourfile
```

在输出结果中，Machine: 后面的值就是 e_machine 字段的值，它指示了 ELF 文件的目标架构。例如，对于 x86-64 架构，这个值通常是 Advanced Micro Devices X86-64。对于其他架构，如 ARM 或 MIPS，这个值会相应地不同。通过查看这个值，你可以确定 ELF 文件是为哪种 ISA 编译的。

在`nanos-lite/src/loader.c`的`static uintptr_t loader(PCB *pcb, const char *filename)`函数中加入一条`if(elf->e_machine!=EXPECT_TYPE) panic("您可能因为疏忽, 让native的Nanos-lite来加载运行一个x86/mips32/riscv32的dummy.");`语句来检测ELF文件的ISA类型，避免让native的Nanos-lite来加载运行一个dummy

对于批处理系统来说, 系统调用是必须的吗? 如果直接把AM的API暴露给批处理系统中的程序, 会不会有问题呢?

对于批处理系统来说，系统调用（system calls）是否必须取决于批处理系统的设计和功能需求。

系统调用是操作系统提供给用户空间程序的一种接口，允许程序请求操作系统的服务。这些服务包括文件操作、进程控制、通信等。

我们考虑一下批处理系统的需求：

如果批处理系统只需要执行一些简单的、预定的任务，并且这些任务不需要操作系统的高级服务，那么可能不需要系统调用；如果批处理系统需要进行文件操作、进程管理或其他需要操作系统支持的操作，那么系统调用是必须的。

直接将抽象机（AM）的API暴露给批处理系统中的程序可能会带来一些问题：

1. 安全性：系统调用提供了一种受控的方式来访问操作系统服务，直接暴露AM的API可能会绕过这些安全控制。
2. 稳定性：系统调用通常经过了严格的测试和优化，以确保操作系统的稳定性。直接使用AM的API可能会引入不稳定因素。
3. 可移植性：系统调用为程序提供了一个与硬件和具体实现无关的接口。直接依赖AM的API可能会降低程序的可移植性。
4. 维护性：如果AM的API直接暴露给程序，那么任何AM内部的更改都可能影响到依赖这些API的程序，增加了维护的复杂性。

一些替代方案：

1. 封装AM的API：可以通过系统调用封装AM的API，这样既可以提供必要的服务，又可以保持操作系统的控制和安全性。
2. 使用库函数：可以提供一组库函数作为AM API和系统调用之间的抽象层，这样程序可以通过这些库函数间接访问AM的服务。

总结来说，系统调用对于需要操作系统服务的批处理系统是必要的。直接将AM的API暴露给程序可能会带来安全、稳定性和维护性方面的问题。通常，通过系统调用来提供服务是一种更安全、更可控的方法。

RISC-V系统调用号的传递，没有通过a0传递系统调用号

在 RISC-V 架构中，系统调用号的传递并没有统一的标准，不同的操作系统和实现可能会有不同的约定。对于 RISC-V Linux 来说，它并没有使用 a0 寄存器来传递系统调用号，而是选择了其他寄存器。这种选择可能是基于以下几个考虑：

1. 函数调用约定：根据 RISC-V 的函数调用约定，a0-a7 这八个寄存器被用来传递函数参数，其中 a0 和 a1 也用来返回值传递。在系统调用中，如果使用 a0 来传递系统调用号，那么第一个参数将需要使用 a1 传递，这可能会与函数调用约定产生冲突，特别是在系统调用和函数调用共用同一套寄存器时。
2. 参数传递效率：RISC-V 调用约定中，参数尽可能使用寄存器传递，以提高效率。如果系统调用号占据了 a0，那么在传递参数时，原本可以通过 a0 传递的参数就需要使用其他寄存器或栈，这可能会降低参数传递的效率。
3. 寄存器使用灵活性：在 RISC-V 中，a7 被用作线程指针（thread pointer），而在某些上下文中，它可能不经常用于普通的函数参数传递。因此，使用 a7 作为系统调用号的传递寄存器，可以避免与函数参数传递的冲突，同时保持寄存器使用的灵活性。
4. 系统调用实现：在 RISC-V Linux 中，系统调用是通过 ecall 指令实现的，ecall 会产生一个异常，scause 中体现为 Environment call from U-mode。系统调用号的传递需要一个寄存器，而 a7 作为系统调用号的传递寄存器，可以方便地与 ecall 指令配合使用，实现系统调用的识别和参数传递。

综上所述，RISC-V Linux 没有使用 a0 来传递系统调用号，而是选择了 a7，这可能是为了与函数调用约定保持一致，提高参数传递效率，以及保持寄存器使用的灵活性。这样的设计也有助于区分系统调用和普通函数调用，简化系统调用的实现。

支持多个ELF的ftrace

在`nemu/src/utils/ftrace.c`中的函数`void print_func_name(const char *elf_file)`又parse解析elf文件，这里可以进行相关的实现让NEMU的ftrace支持多个ELF文件，可以使用一个for循环遍历多个ELF文件，但是因为个人比较懒所以暂且先不实现了吧。

用fopen()还是open()？

应当使用open()函数，`int fd = open("/dev/events", 0 , 0);`

- open是UNIX系统调用函数（包括LINUX等），返回的是文件描述符（File Descriptor），它是文件在文件描述符表里的索引。
- fopen是ANSIC标准中的C语言库函数，在不同的系统中应该调用不同的内核api。返回的是一个指向文件结构的指针。

open返回文件描述符，而文件描述符是UNIX系统下的一个重要概念，UNIX下的一切设备都是以文件的形式操作。如网络套接字、硬件设备等。当然包括操作普通正规文件（Regular File）。open函数运行在内核态，离系统内核更近 fopen是用来操纵普通正规文件（Regular File）的。

fopen()函数与open()函数区别在于fopen属于缓冲文件系统，fopen, fclose, fread, fwrite, fseek等都有缓冲区，而open属于非缓冲文件系统，相关文件操作的函数有open, close, read, write, lseek，由于键盘是字符设备，而且写入速度（用户键盘输入）十分慢，不需要缓冲区，因此选择后者

一句话总结一下，就是open无缓冲，fopen有缓冲。前者与read, write等配合使用， 后者与fread,fwrite等配合使用。

比较fixedpt和float，用fixedpt来模拟表示float, 这其中隐含着哪些取舍?

- 首先就是对于精度的取舍，定点数的精度受限于小数位的位数。例如，在“24.8”格式中，最小的表示单位是 $\frac{1}{256}$ ，这意味着不能表示比这个更精细的值。与浮点数相比，定点数的精度较低，尤其是对于非常大或非常小的数。

- 另外取值范围也有取舍，定点数的表示范围会比浮点数小得多。如果需要表示超出定点数范围的数，就需要改变比例因子或使用更大的数据类型，这可能会进一步限制小数部分的精度。

- 在运算层面，定点数的加、减、乘、除运算比浮点数简单，因为它们可以映射到整数运算。但是，这也意味着需要手动处理比例因子，可能会引入额外的开销。

- 对于特殊值的处理，浮点数有特殊的值，如无穷大、非数（NaN）和零，这些在定点数中可能没有直接的表示方法。标准化和兼容性问题，浮点数遵循IEEE标准，被广泛支持。而定点数没有统一的标准，可能需要自定义实现，这可能导致兼容性问题。

阅读fixedpt_rconst()的代码, 从表面上看, 它带有非常明显的浮点操作, 但从编译结果来看却没有任何浮点指令. 你知道其中的原因吗?

fixedpt_rconst() 函数的名称暗示它是用来定义一个固定的定点常量。从代码中可以看出，这个函数并不是真正的函数调用，而是宏或者模板，它在编译时将浮点数常量转换为定点数表示。

定点数（Fixed-point number）是计算机编程中用来近似浮点数算术的一种方法，它将小数部分固定在整数的某个区间内。这种方法避免了使用浮点硬件或浮点类型的开销，同时也可以减少浮点运算的精度误差。

在这段代码中，fixedpt_rconst() 被定义为一个宏，它在编译时将浮点数转换为定点数。这意味着所有的浮点运算实际上在编译时就已经被转换为定点数运算，并且在编译结果中只包含整数运算指令。

例如，fixedpt_rconst() 宏定义如下：

```
#define FIXEDPT_BITS 32
#define FIXEDPT_FBITS 8
#define FIXEDPT_ONE (1 << FIXEDPT_FBITS)

#define fixedpt_rconst(x) ((fixedpt)((x) * FIXEDPT_ONE))
```

在这个宏中，x 是一个浮点数，FIXEDPT_ONE 是定点数中的 "1.0" 表示。通过乘以 FIXEDPT_ONE，浮点数 x 被转换为对应的定点数表示。这个转换是在编译时完成的，因此编译后的代码中不包含任何浮点指令。

由于所有的浮点数都被转换为定点数，并且在编译时就已经确定了它们的值，编译器可以优化掉所有的浮点运算，只保留必要的整数运算指令。这就是为什么从编译结果来看没有任何浮点指令的原因。所有的浮点运算都被预先计算并转换为定点数运算，这使得程序可以在不支持浮点硬件的环境下运行，同时也可以减少运行时的计算误差。

如何将浮点变量转换成fixedpt类型?

在不引入浮点指令的情况下将浮点变量转换成 fixedpt 类型，我们需要了解浮点数和定点数在内存中的表示方式。对于一个32位的浮点数（通常遵循IEEE 754标准），它由1位符号位、8位指数位和23位尾数（有效数字）组成。

将浮点数转化为定点数可以执行：

1. 读取浮点数的尾数：忽略指数位和符号位，只取尾数部分。
2. 调整尾数：将尾数左移23位，使其成为一个整数。
3. 考虑符号：如果原始浮点数是负数，确保结果也是负数。
4. 缩放：根据 fixedpt 类型的精度，可能需要将整数缩放（乘以或除以）以适应定点数的范围。

一些实验日志

自陷指令可以认为是一种特殊的无条件失败

对于riscv32来说执行ecall自陷指令时会提供mtvec寄存器来存放异常入口地址. 为了保存程序当前的状态, riscv32提供了一些特殊的系统寄存器, 叫控制状态寄存器(CSR寄存器) 在PA中, 我们只使用如下3个CSR寄存器:

1. mepc寄存器 - 存放触发异常的PC
2. mstatus寄存器 - 存放处理器的状态
3. mcause寄存器 - 存放触发异常的原因

当riscv32触发异常之后硬件过程如下：

1. 将当前PC值保存到mepc寄存器（即存放触发异常的PC）
2. 在mcause寄存器中设置异常号（即存放触发异常的原因）
3. 从mtvec寄存器中取出异常入口地址
4. 跳转到异常入口地址

若决定无需杀死当前程序, 等到异常处理结束之后, 就根据之前保存的信息恢复程序的状态, 并从异常处理过程中返回到程序触发异常之前的状态. riscv32通过mret指令从异常处理过程中返回, 它将根据mepc寄存器恢复PC.

设置异常入口地址

当我们选择yield test时, am-tests会通过cte_init()函数对CTE进行初始化, 其中包含一些简单的宏展开代码. 这最终会调用位于abstract-machine/am/src/$ISA/nemu/cte.c中的cte_init()函数. cte_init()函数会做两件事情, 第一件就是设置异常入口地址:对于riscv32来说, 直接将异常入口地址设置到mtvec寄存器中即可. cte_init()函数做的第二件事是注册一个事件处理回调函数, 这个回调函数由yield test提供

对于控制状态寄存器的相关指令补充

```
csrrw命令：写控制状态寄存器  对于x[rs1]中每一个为 1 的位，把控制状态寄存器 csr 的的对应
位置位，等同于 csrrs x0, csr, rs1.
800013b0:   30571073            csrrw   mtvec,a4        001100000101 01110
001 00000 1110011
对于csrrs命令：记控制状态寄存器 csr 中的值为 t。把 t 和寄存器 x[rs1] 按位或的结果写入
csr，再把 t 写入x[rd]。
80001454:       342022f3        csrr    t0,mcause   001101000010 00000 010
00101 1110011
```

进入异常入口地址

需要在自陷指令(我将其定义为ECALL宏)的实现中调用isa_raise_intr(), 而不要把异常响应机制的代码放在自陷指令的helper函数中实现, 因为在后面我们会再次用到isa_raise_intr()函数. 实现isa_raise_intr，这个函数其实就是模拟了ecall的功能，即使得程序跳转到异常处理中，根据注释提示我们知道NO对应异常种类，epc对应触发异常的指令地址，最后返回异常入口地址

```
word_t isa_raise_intr(word_t NO, vaddr_t epc) {
  cpu.csr.mcause = NO;
  cpu.csr.mepc = epc;
  return cpu.csr.mtvec;
}
```

保存上下文

上下文信息包括： 通用寄存器组 触发异常时的PC和处理器状态 异常号（riscv32的异常号已经由硬件保存在mcause寄存器中, 我们还需要将其保存在堆栈上.） 地址空间

这里我出了一个bug，由于riscv32中的异常处理过程到了最后mret的时候是继续执行还是回到最初的样子是不一样的，我一开始没有想那么多，只是依照着riscv32的指令集手册去完成指令，结果最后一直输出大量的y，之后继续往下做才发现自己的问题……已经在nemu/src/isa/riscv32/system/intr.c中加上一个判断条件即可

恢复上下文最重要的是恢复执行流程，即恢复pc，触发中断的指令地址保存在mepc中，那么mret就负责从用这个寄存器恢复pc。需要注意的是自陷只是其中一种异常类型. 有一种故障类异常，例如缺页异常, 在系统将故障排除后, 将会重新执行相同的指令进行重试, 此异常返回的PC无需加4。所以根据异常类型的不同, 有时候需要加4, 有时候则不需要加，在什么地方做这个+4的决定呢？我们在ecall时判断中断类型并+4即可

Nanos-lite

Nanos-lite和NEMU是两个独立的项目, 它们的代码不会相互影响, 你在阅读代码的时候需要注意这一点。但是nanos-lite里面是可以直接使用abstract-machine里面的接口函数的

navy子项目

为了回答可执行文件在哪里，我们还要先说明一下用户程序是从哪里来的. 用户程序运行在操作系统之上, 由于运行时环境的差异, 我们不能把编译到AM上的程序放到操作系统上运行. 为此, 我们准备了一个新的子项目Navy-apps, 专门用于编译出操作系统的用户程序.

小问题

一开始并没有找到libc这个文件夹，并且在

```
#为了编译dummy, 在navy-apps/tests/dummy/目录下执行
make ISA=$ISA
```

这一块的时候make一直失败显示没有对应的Makefile文件，后来重新关掉shell再重新打开编译就可以了，需要清理工作区，有时候，工作区中的旧文件或不相关的文件可能会导致问题。尝试清理工作区，然后重新运行 make。

差点就要reset了

```
//说一件特别恐怖的事情  如果我在这里使用了_end而不是end的话  最后竟然没有办法成功运行（连dummy都会失败）  想不明白为什么……
//难道又是我电脑出bug了（因为debug的时候发现printf ramdisk里面的RAMDISK_SIZE会为0 但是如果我修改_end为end之后重新makeisa在运行就可以通过了）
//然后还有如果在_end的情况下我printf一会正确一会错误  难道是因为printf改变了程序的状态？太奇怪了……
```

abstract-machine/am/include/arch/riscv.h文件GPR设置出问题

```
//下面这个是写错了的！！！！！！！！！！！！！！！！！！！！可是debug了将近4个小时
// #define GPR2 gpr[4]  //观察_syscall_的代码  asm (GPR2) = a0;  而a0正好为gpr[4]
// #define GPR3 gpr[5]  //asm (GPR3) = a1;  a1正好为gpr[5]
// #define GPR4 gpr[6]  //asm (GPR4) = a2;  a2正好为gpr[6]
// #define GPRx gpr[4]  //  GPRx为a0  观察_syscall_的代码，发现是从a0寄存器取得系统调用的返回结果，因此修改$ISA-nemu.h中GPRx的宏定义，将其改成寄存器a0的下标，然后就可以在操作系统中通过c->GPRx根据实际情况设置返回值了
// 处理系统调用的最后一件事就是设置系统调用的返回值. 对于不同的ISA, 系统调用的返回值存放在不同的寄存器中,
// 宏GPRx用于实现这一抽象, 所以我们通过GPRx来进行设置系统调用返回值即可.

#define GPR2 gpr[10]  //观察_syscall_的代码  asm (GPR2) = a0;  而a0正好为gpr[10]
#define GPR3 gpr[11]  //asm (GPR3) = a1;  a1正好为gpr[11]
#define GPR4 gpr[12]  //asm (GPR4) = a2;  a2正好为gpr[12]
#define GPRx gpr[10]  //  GPRx为a0  观察_syscall_的代码，发现是从a0寄存器取得系统调用的返回结果，因此修改$ISA-nemu.h中GPRx的宏定义，将其改成寄存器a0的下标，然后就可以在操作系统中通过c->GPRx根据实际情况设置返回值了

//完了好像知道自己问题出在哪里了……这里的c->GPR不是navy-apps/libs/libos/src/syscall.c中的GPR宏定义  而是在abstract-machine/am/include/arch/riscv.h中的……shit啊啊啊啊啊啊啊啊啊  我这debug了好久啊啊啊
  //但是不对啊 观察riscv.h中也一样啊……怎么回事啊啊啊啊啊啊
  //啊啊啊啊啊啊啊啊啊啊啊啊啊在debug4个小时之后我终于知道自己错在哪里了啊啊啊啊啊啊  原来是riscv.h中的宏定义我写错了……我……无语了哥们……
```

一切皆文件与VFS虚拟文件系统

为了向用户程序提供统一的抽象, Nanos-lite也尝试将IOE抽象成文件.

我们之前实现的文件操作就需要进行扩展了: 我们不仅需要对普通文件进行读写, 还需要支持各种"特殊文件"的操作. 至于扩展的方式, 你是再熟悉不过的了, 那就是抽象!我们对之前实现的文件操作API的语义进行扩展, 让它们可以支持任意文件(包括"特殊文件")的操作:

```
int fs_open(const char *pathname, int flags, int mode);
size_t fs_read(int fd, void *buf, size_t len);
size_t fs_write(int fd, const void *buf, size_t len);
size_t fs_lseek(int fd, size_t offset, int whence);
int fs_close(int fd);
```

再次遇见了debug了一天的错误

```
}
size_t fs_write(int fd, const void *buf, size_t len)
{
  // if(fd==FD_STDIN){Log("忽略此次写入标准输入文件%s", file_table[fd].name);}
  //除了写入stdout和stderr之外(用putch()输出到串口), 其余对于stdin, stdout和stderr这三个特殊文件的操作可以直接忽略
  // if(fd==FD_STDOUT||fd==FD_STDERR){for(size_t i=0;i<len;i++){putch(*( (char*)buf+i) );}return len;} //如果是标准输出或者标
  WriteFn writefun=file_table[fd].write;
  //##################################################################################################################
  // if(writefun!=NULL)return writefun(buf,0,len);
  if(writefun!=NULL){ if(writefun==serial_write) return serial_write(buf,0,len);
    else if (writefun==fb_write) return fb_write(buf,file_table[fd].disk_offset+file_table[fd].open_offset,len);
    //啊啊啊啊啊这个地方一开始卡了好久  因为默认非NULL的时候offset为0……无语死了……
    else return writefun(buf,0,len);}
  //##################################################################################################################
  size_t write_len=len;//接下来的过程和fs_read几乎一模一样了
```

一直有一个问题卡了我将近1天啊啊啊啊啊啊啊啊，又是因为没有好好RTFSC的问题，我当时看手册的时候 没有多想 有一句说约定“函数指针为NULL时, 表示相应文件是一个普通文件” 然后我就想着那默认offset的值应该是0吧

然后结果一下子就在if语句里面把offset都默认为0，结果debug的时候在NDL当中的offset是正常的，结果编译到riscv32-nemu上去了之后offset竟然就变成0了……当时真的百思不得其解，甚至有想过是不是系统调用少了，然后再去检查fs_write的部分发现竟然是这个问题……吃大亏了呀……我哭死……

无语死了 没有好好看手册 导致文件夹名字没有改 debug了好久 但是也是学会使用gdb了哈 记得在makefile当中的CFLAG+=-g将release版本变为debug版本

真的没绷住 差点PA3重开了 因为ramdisk.img的大小不能超过48MB，然后我当时nslider里面的大小太大了 导致超出范围 一直报错报错啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊啊 这可是de了一天的bug啊 差点人就以头抢地了……

在navy-apps/libs/libminiSDL/src/video.c中的SDL_UpdateRect函数当中需要有:

```
// // 仙剑奇侠传中的像素阵列存放的是8位的调色板下标,
// // 用这个下标在调色板中进行索引, 得到的才是32位的颜色信息
// uint32_t pal_color_xy = palette[pixels[x][y]];
  static uint32_t trans_color_from_8_to_32(SDL_Color *c)
  {return (c->a << 24) | (c->r << 16) | (c->g << 8) | c->b;//将8位color转换成
32位color
  }
```

```
  local_pixels[i * w + j]=trans_color_from_8_to_32(&colors[pixel_index]);
  //SDL_UpdateRect()在最后调用NDL_DrawRect()，传入的参数pixels必须是32位格式的，不然会显示错误颜色。
  //因此需要现将8位颜色转为32位的，再填入pixels
```

后面正常运行之后前面就无法正常运行了

不知道为什么在minisdl的实现过程中 void SDL_UpdateRect(SDL_Surface *s, int x, int y, int w, int h) 这个函数一直出问题，我尝试过case转换成if去做但是还是有问题，导致不能分类讨论，我个人目前暂且认为是因为编译器优化问题然后最后使得map函数assert0了……报错原因是0x0000004 out of bound