Nemu攻略（南大ics2015）

Nemu 是一个功能完备但经过简化的模拟器，该项目有2021年的最新版本，但是本笔记攻略的是ics2015。笔记中记录了ics2015中除了TLB实现、DMA实现的全部内容。最后在模拟器上运行一个小游戏，本人并未成功运行 仙剑奇侠传 ，但是对应的实验内容已经完成（可能是本人书写代码中含有bug😁），但是考虑到这个项目已经不在维护，不再深入探讨仙剑无法运行的原因。实验分为四部分:

- [x] 简易调试器
- [x] 指令系统
- [x] 存储管理
- [x] 中断与I/O

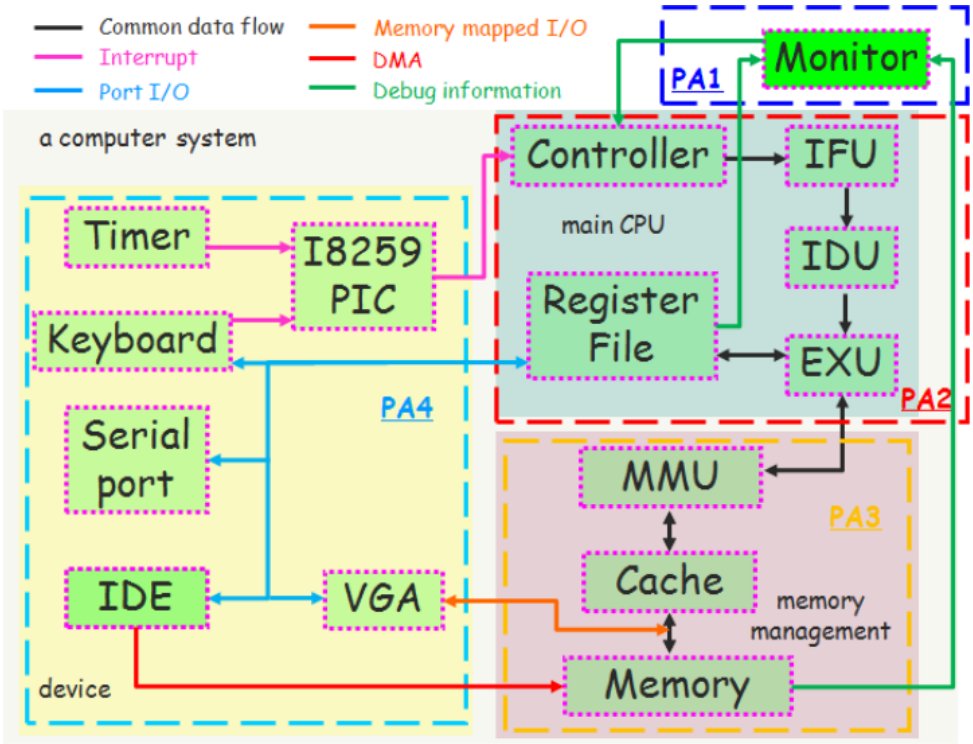
实验准备

```
* virtualBox6.1
* ubuntu18.04
* gcc 7.50
* sdl1.2
```

在配置完这些后可以获取代码，并安装必要的库

```
git clone https://github.com/NJU-ProjectN/ics2015
apt-get install libreadline-dev
apt-get install libsdl1.2-dev
```

PA1 简易调试器



把计算机理解为一个状态机的话，那么Nemu也是要实现一个状态机，Monitor就是用户和这个状态机交互的接口，让用户可以实时了解Nemu所处的状态并控制Nemu。特别是在指令的实现过程中，Nemu的调试功能将发挥巨大的作用。

```
ics2015
├── config           # 包含Makefile的一些配置，包含Makefile公用的函数
├── game             # 包含打字小游戏和仙剑奇侠传两个游戏
├── kernel           # 微型操作系统内核
├── lib-common       # 公用的库
├── Makefile         # 提供工程的构建，运行，测试，打包等功能
├── nemu             # NEMU
├── testcase         # 用C写的测试用例
└── test.sh          # 实现到内核的部分后，开启自动测试的测试脚本
```

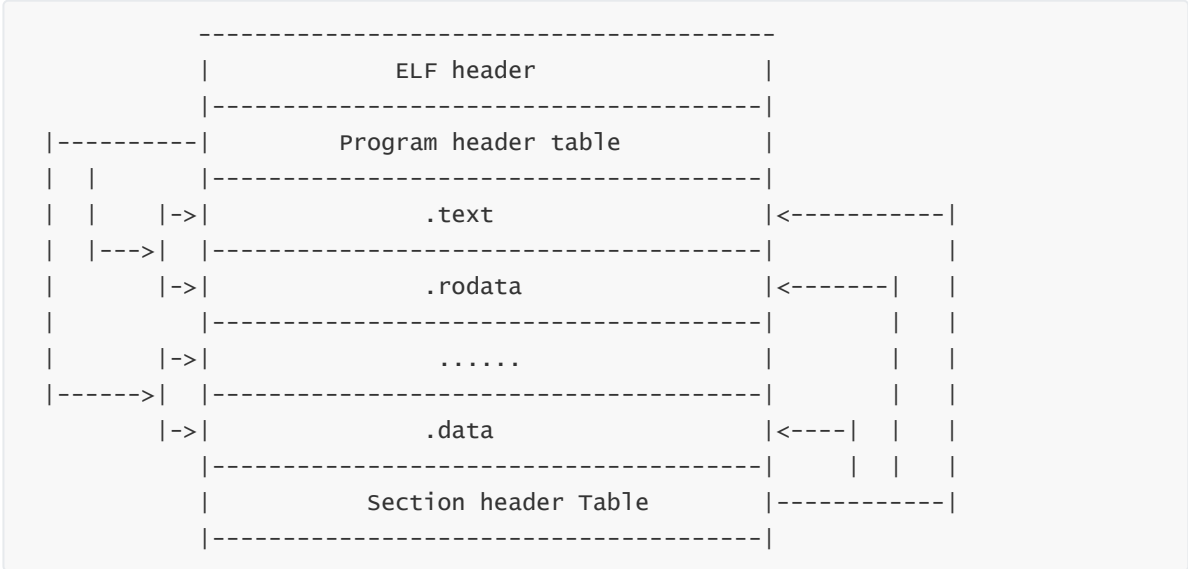
PA1中首先要求你先大致了解Nemu子目录结构，但是对于什么都不知道的学习者，即使你说了很简洁凝练的概括，也只能是听个热闹。不如找一个点切入，这里从 *main.c* 开始。

```
\\ nemu/src/main.c

int main(int argc, char *argv[])
{
    //nemu的参数显然是要执行的程序，可以预见init_monitor的作用
    init_monitor(argc, argv);

    //测试寄存器的实现是否出现了问题
    reg_test();

    //硬件模拟的初始化
    restart();

    //进入Debugger（就是monitor）主循环
    ui_mainloop();
}

\\ nemu/src/monitor/monitor.c
init_monitor(int argc, char *argv[])
{
    //日志文件，nemu每执行一条指令都会留下记录
    init_log();

    //加载符号表
    load_elf_tables(argc, argv);

    //用于对debugger命令的解析
    init_regex();

    //链状结构，实现对watchpoint的管理
    init_wp_pool();

    welcome();
}
```

init_monitor中执行了load_elf_table，对elf的解析是nemu中一个很关键的部分，在PA4里也常常需要去理解ELF文件的格式。

ELF文件解析

在ELF文件中，elf header后面有一系列的section，有几个section又组成了segment，为了描述它们，就分别需要section table和segment table（Program header table）。在load_elf_table中要做的任务就是从section table中找到符号表的信息。首先Elf header中有section table的偏移和section对应的名字表的偏移，section table entry每一个中的sh_name中指明了自己名字在名字表中的偏移。通过section table这样就可以找到符号表了，当然符号表的名字信息存在string table中，它也在section table可以找到信息。




```
\\ ics2015\nemu\src\monitor\debug\elf.c
void load_elf_tables(int argc, char *argv[])
{
    exec_file = argv[1];

    File *fp = fopen(exec_file,"rb");

    uint8_t buf[sizeof(Elf32_Ehdr)];
    ret = fread(buf, sizeof(Elf32_Ehdr), 1, fp);

    Elf32_Ehdr *elf = (void *)buf; //把二进制内容当作elf header解析

    //中间有很多内容不关键,对assert内容进行省略

    uint32_t sh_size = elf->e_shentsize * elf->e_shum; //获取section table大小
    Elf32_Shdr *sh = malloc(sh_size);
    fseek(fp, elf->e_shoff, SEEK_SET);//e_shoff 意味着fp指针现在指向了section table
    ret = fread(sh, sh_size, 1, fp); //section table被读入了sh

   //下面同理读入section header string table,里面记录了section的名字，用section table中的sh_name作为下标可以直接访问section的名字。
    char * shstrtab = malloc(sh[elf->e_shstrndx].sh_size);
    fseek(fp, sh[elf->e_shstrndx]);
    ret = fread(shstrtab, sh[elf->e_shstrndx].sh_size, 1, fp);

    //遍历section table找到section table中type为SHT_TAB和SHT_STRTB的entry，同时也确实它们在section header string table中名字分别为".symtab"和“.strtab"

  ......
}
```

那么有了符号表可以做什么呢，答案是对函数中用到的全局变量和函数进行解析，它们都可以在符号表中找到。Nemu中实现了两个函数：

```
//这个函数用于查找名字为name的全局变量的虚拟地址，symbol table第i项的名字下标为stmtab[i].st_name,它存在string table中。如过名字相同，说明找到了，st_value存着对应的虚拟地址
int findSym(char *name)
{
    int i;
    for(i = 0; i < nr_symtab_entry; i++ )
    {
        if( strcmp( &strtab[symtab[i].st_name], name ) == 0 )
            return symtab[i].st_value;
    }
}

//这个函数用于查找addr所在的函数地址
int findFunc(uint32_t addr)
{
    for(i=0;i<nr_symtab_entry;i++)
    {
        if(addr >= symtab[i].st_value && addr < symtab[i].st_value + symtab[i].st_size)
                        return &strtab[symtab[i].st_name];
    }
        return NULL;

}
```

正则表达式

解析完elf，在看regex的内容，它使用系统的库用于解析命令为token。init_regex中我们调用regcomp对用到的规则进行编译。如 `==` , `0[xX][0-9a-fA-F]` 等等, 这些规则被识别出来后，就认定这个字符串是某一个类型的token，就如0x09就被前面提到的规则识别为16进制数。下面是一个例子:

```
p 1+2 ---->识别出是p命令------>解析1+2-------->
token(1,NUM),token(+,ADD),token(2,NUM)
```

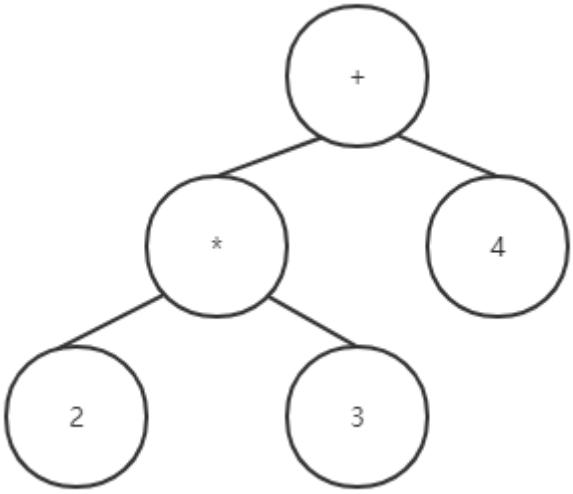
我们可以这样理解Nemu中debugger运行逻辑，我们总将第一个单词识别为命令，之后所有的字符串都将作为参数交给函数expr,解析如下：

```
uint32_t expr(char *e, bool *success)
{
    //make_token识别参数为token串
    if( !make_token(e) )
    {
        *success = false;
        return 0;
    }

    //下面这个循环用寻找token中的*,因为我们再词法分析阶段无法知道这个*到底是乘号还是取指针指向的值，判断的标准就是*前是否是操作符，乘号前不能是操作符的，当然这个判断方法存在错误的可能
    int i;
    for( i=0; i < nr_token; i++ )
    {
```

```
        if(tokens[i].type == '*' && (i == 0 || operator_judge(tokens[i -
1].type) ) )
            tokens[i].type = DRREF;
    }

    //中缀树求值
    int result = eval( 0, nr_token - 1, success);
    return result;
}
```

而下面分为两个部分表达式解析为token的具体过程和中缀树求值的过程。

```
while(e[position] != '\0')
{
       /* Try all rules one by one. */
       //在regex中我们对所有规则进行了编译，re存了编译后的结果，对字符串e + position 进行匹
配，结果存到pmatch中
       //这里rules数组和re数组一一对应，re的第i项匹配成功，说明了它的token类型就是
rules[i].token_type，将得到的token存储到token序列
       for(i = 0; i < NR_REGEX; i ++)
       {
            if(regexec(&re[i], e + position, 1, &pmatch, 0) == 0 && pmatch.rm_so
== 0)
            {
                char *substr_start = e + position;
                int substr_len = pmatch.rm_eo;
                position += substr_len;

                switch(rules[i].token_type)
                {
                    case NOTYPE:
                        break;
                    case HEX:
                    case NUM:
                    case REG:
                    case SYMB:
                        //如果为数据的话要存入数据的字符，如$eax来在求值时使用
                        strncpy(tokens[nr_token].str,substr_start,substr_len);
                        tokens[nr_token].str[substr_len]='\0';
                    default:
                        tokens[nr_token].type = rules[i].token_type;
                        nr_token++;
                }

                break;
            }
        }

    //错误
    ......
}
```



中缀树如图所示，对于一个表达式，可以通过一个运算符分成左右两个部分，左右两个部分又是一个中缀树于是就可以递归运算直至只有一个节点。

这个eval函数就是用于解析这个过程，函数本身有许多细节不再考虑，我们讲讲它的原理：
* int eval(int left, int right, bool& success)。left和right代表了token数组的index，left到right就代表了要解析的表达式的范围，success用记录解析过程中是否发生了错误。
* 递归的出口,当right == left说明子表达式只有一个token，那么如果表达式本身没有问题，那么它应该是一个操作数，根据tokens[left]中的str可以知道它所代表的值，如$eax就是cpu.eax的值。
* right < left，这是发生了错误,直接回退。
* 如果left是一个'('，right是一个')',我们可以直接调用eval(left+1,right-1,success)
* left<right时，我们需要找到一个运算符使得整个表达式分为左右两个部分，根据中缀树的性质，这个运算符在整个表达式中应该最后运算，所以我们应该找到最低优先级的运算发。于是我们从右往左遍历，设置一个prior = 15，去找是否有15优先级的运算符，没有的话 prior-- 再试一次。找到的话，就把这个运算符作为dominator，以它为界划分为左右两个。

watchpoint

watchpoint用于检查一个表达式的值是否发生了变化，如$eax，如果eax寄存器的值发生了变化那么nemu就会停止运行，为我们的调试提供便利。watchpoint的实现原理如下，我们申请了32个空闲watchpoint池用于管理空闲的watchpoint，用一个free指针指向空闲池的头部，每次申请从头部去除一个watchpoint最为head队列的头部。于是我们就有了两个队列free队列（用于管理空闲的watchpoint资源），head队列（用于管理已经申请到的watchpoint资源）。每次指令执行完成之后，我们都执行一次check_wp(), 遍历一次head队列，检查一次wp的值是否和初始的值一样，如果不一样nemu就会停止执行，等待我们输入新的命令。至此，我们已经分析完了init_monitor的内容，下面的加载可执行文件就是restart的内容。

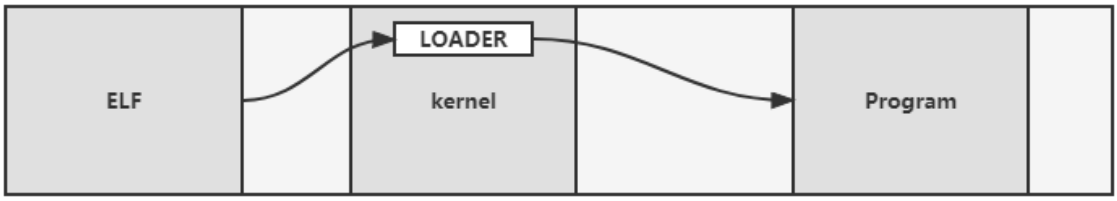
加载可执行文件

再看restart, 它依次执行了init_ramdisk、load_entry、init_ddr3、init_cache、init_L2、init_device、init_sdl。最后的项目中不再使用ramdisk，因为我们实现了nemu中的磁盘模拟，但是其实原理是一模一样的，最后没有模拟IDE，我们仍然可以通过ramdisk实现相同的功能。

```
//可以看见我们将输入的ELF文件加载到了模拟的内存数组中
static void init_ramdisk()
{
    int ret;
    const int ramdisk_max_size = 0xa0000;
    FILE *fp = fopen(exec_file, "rb");
```

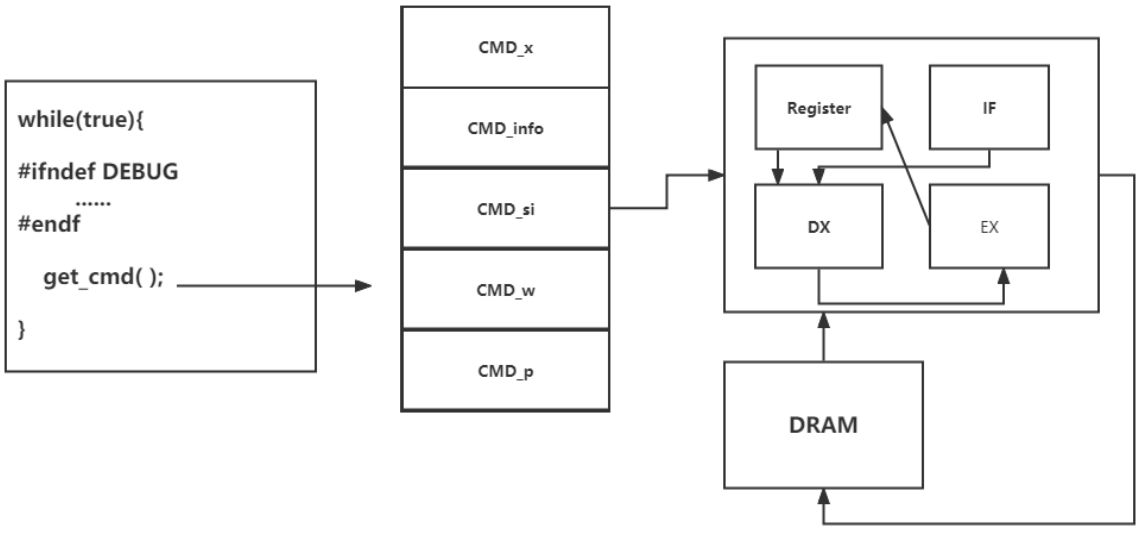
```
    fseek(fp, 0, SEEK_END);
    size_t file_size = ftell(fp);

    fseek(fp, 0, SEEK_SET);
    ret = fread(hwa_to_va(0), file_size, 1, fp);

    fclose(fp);
}
```

```
//entry文件被加载到了模拟内存的ENTRY_START（0x100000）
static void load_entry() {
        int ret;
        FILE *fp = fopen("entry", "rb");
        Assert(fp, "Can not open 'entry'");

        fseek(fp, 0, SEEK_END);
        size_t file_size = ftell(fp);

        fseek(fp, 0, SEEK_SET);
        ret = fread(hwa_to_va(ENTRY_START), file_size, 1, fp);
        assert(ret == 1);
        fclose(fp);
}
```

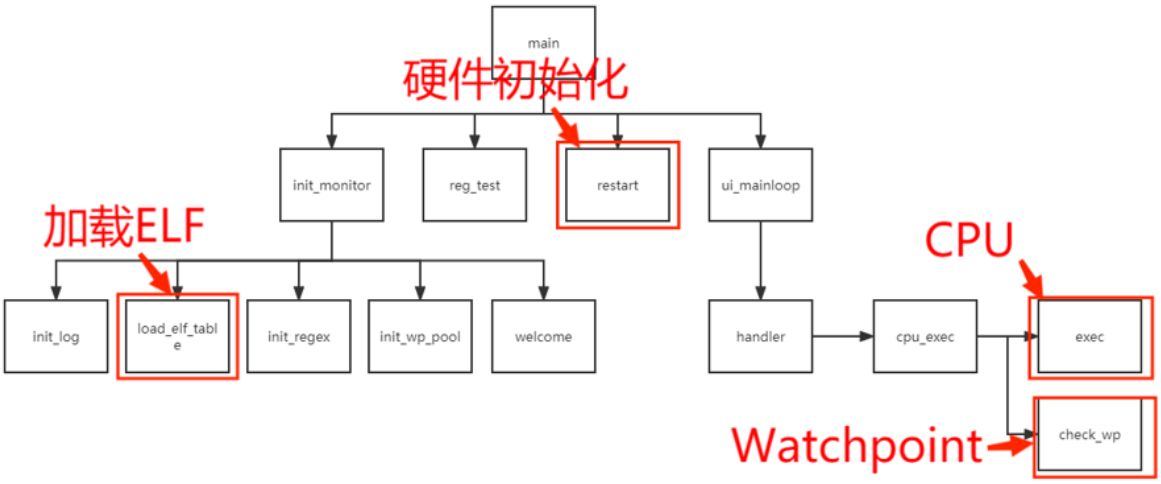


我们现在没有kernel，所以entry和elf我们都加载编译的testcase文件，我们一开始在kernel的位置0x100000开始执行。于是我们就可以通过nemu中执行32位文件。

交互界面



很直观的理解，交互界面是一个死循环，每次通过get_cmd获得一个新的命令，去解析。PA1中我们只用实现cmd_x(内存扫描)、cmd_info(信息显示)、cmd_si（单步执行)、cmd_w(设置watchpoint)、cmd_p(表达式求值)这个几个功能，当然有的命令已经在框架代码中实现了。我们通过readline读取命令，strtok隔开空格，第一个空格前是命令，空格后是参数。根据命令执行cmd_info、cmd_x、cmd_si等等。



分析到这里我们已经知道了debugger的框架，main函数中分别执行了monitor初始化，硬件初始化，跳入debugger循环。其中monitor初始化中load_elf_table加载了ELF符号表，restart中加载文件到我们模拟的内存数组中并进行了硬件部分的初始化，最后跳入了交互循环，根据输入执行handler函数，handler函数中最为关键的就是exec，PA2中我们将通过实现它完成对一条指令的执行，而每次执行完一条指令我们执行check_wp来检查是否触发了watchpoint。最后补充，不要将cpu_exec和exec搞混，cpu_exec中调用了一个循环，循环调用exec，并输出刚刚执行的指令，最后检查watchpoint，严格来说属于monitor的一部分（实时上也是），exec属于cpu的一部分，它收到一个eip执行地址在eip处的指令，也就是CPU取指、译码、执行的工作。

PA2 指令系统

为什么学习计算机系统

这一节本来不在PA2中，而是PA整个实验的一个附属内容，也是计算机系统课程的开头。但是这一节的问题，对于实现IA32的指令执行具有指导性的意义，也是csapp学习中的核心问题。

1.数组求和

```
//为什么当len为0时，会触发Access Violation
int sum(int a[ ], unsigned len) {
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

C中通过32位bit存储unsigned int和int，而无符号值最小值为0，当unsigned int的0减去1，结果将会是一个巨大的正数

2.整数的平方

若 x 和 y 为 int 型，当 x = 65535 时，则 y = x*x = -131071 . 为什么?

因为int为32为补码，65536是2^16,65536*65536就是2^32，就是符号位为1，其他为0，于是发生了溢出，即超出了int的最大表示

3.多重定义符号

```
/*---main.c---*/
#include <stdio.h>
int d=100;
int x=200;
void p1(void);
int main() {
    p1();
    printf("d=%d,x=%d\n",d,x);
    return 0;
}

/*---p1.c---*/
double d;
void p1() {
    d=1.0;
}
```

在上述两个模块链接生成的可执行文件被执行时，main 函数的 printf 语句打印出来的值是:
d=0,x=1072693248 . 为什么不是 d=100,x=200 ?

对于一个全局变量的赋值将别认为是一个强定义，弱定义在强定义面前是无效的，于是在链接时，p1对d的赋值实际上是对main中的d进行赋值，double是8字节数于是d和x都被覆盖。而根据IEEE754来看一个double的1是0x3ff0,0000,0000,0000。于是d被赋值为0，x被赋值为0x3ff00000，而0x3ff00000根据int来解释就是1072693248。

4.奇怪的函数返回值

```
double fun(int i) {
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

从 fun 函数的源码来看，每次返回的值应该都是 3.14 ，可是执行 fun 函数后发现其结果是:
* fun(0) 和 fun(1) 为 3.14
* fun(2) 为 3.1399998664856
* fun(3) 为 2.00000061035156
* fun(4) 为 3.14 并会发生 访问违例 这是为什么?

首先volatile保证了栈中的结构确实如同我们定义的一样是一个double之后跟上long int。我们先看fun(0)正常返回3.14，func(1)即a[1]正常赋值也正常返回3.14。而func(2)发生了越界，a[2]实际上已经不再a数组的范围内，但是d[1]实际上在栈中的空间就是a[2]，double现在的值就是1073741824即0x40000000,3.14的高32位加上低32位的0x40000000就是新的d[0]。而a[3]就是将高位替换为0x40000000。

5.时间复杂度和功能都相同的程序

```
void copyij(int src[2048][2048], int dst[2048][2048]) {
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
void copyji(int src[2048][2048], int dst[2048][2048]) {
      int i,j;
      for (j = 0; j < 2048; j++)
          for (i = 0; i < 2048; i++)
              dst[i][j] = src[i][j];
}
```

上述两个功能完全相同的函数，时间复杂度也完全一样，但在Pentium 4处理器上执行时，所测时间相差大约21倍. 这是为什么? 猜猜看是 copyij 更快还是 copyji 更快?

copyij更块，因为一个二维的矩阵相同的行是相邻存储的，所以copyij在一个行的相邻列访问的时候cache都是命中的，cache的命中使得第一个copy的时间更短。

6.网友贴出的一道百度招聘题

```
#include <stdio.h>
int main() {
    double a = 10;
    printf("a = %d\n", a);
    return 0;
}
```

该程序在IA-32上运行时，打印结果为 a=0 ；在x86-64上运行时，打印出来的 a 是一个不确定值. 为什么?

在x86-64系统上调用的规则和IA32上不同，当使用浮点数的时候使用的是xmm向量寄存器。printf中我们把数据地址输入向量寄存器xmm0,输入格式的字符串地址输入edi，再输入eax向量寄存器的个数为1，之后调用printf，但是由于printf中的格式为%d,实际上取数据是在esi中取参数打印，这时esi中是一个不确定的值。

但是回答IA-32系统上的调用规则，我们使用栈来传递参数，x87浮点指令是先把a从局部变量区取出来压入浮点栈，之后弹入到参数区，由于右参数先入栈，实际上栈中的结构由高到低为a的高位，a的地位，格式化字符串地址，但是由于格式化字符串栈中为%d，所以输出低位。

7.整数除法

- 代码段1

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

- 代码段2

```
int a = 0x80000000;
int b = -1;
int c = a / b;
printf("%d\n", c);
```

> 代码段一的运行结果为 `-2147483648` ; 而代码段二的运行结果为 `Floating point exception` , 显然, 代码段二运行时被检测到了"溢出"异常. 看似同样功能的程序为什么结果完全不同?
>
> 对于代码段1，编译器在生成二进制代码时会直接采用取补码的进行优化，所以0x80000000的补码仍然是0x80000000，所以结果是-2147483648。但是对于代码段2，由于2147383648已经超出int能表示的范围所以发生了溢出。

从Makefile中理解项目结构

在理解了monitor的代码后，可以理解Nemu大致的运行的原理，就是由最上层的debugger进行操作，将指令一条一条地交给cpu执行。cpu解析这部分由多个部分组成，包含地关系比较复杂，我们可以通过Makefile看看项目是如何组织的，从而更好地进行下一步。

```
# prototype: make_command(program, flags, msg, input)
# 用于在满足依赖关系时生成目标文件所需要的指令
define make_command
@echo + $(3)
@mkdir -p $(@D)
@$(1) -o $@ $(4) $(2)
endef

# prototype: make_common_rules(target, cflags_extra)
# 这个函数用于生成eval所需的执行命令的文本，通过函数可以知道一个目标，它的源文件在src目录下，头文件在include目录下，生成的可执行文件在obj对应的目录下。所有所需的源文件都是.c或者.s文件，而且每一个源文件都会生成一个对应.o文件。接下来FLAGS参数指明编译的参数，值得注意的是，汇编编译时这里已经显示地说明了头文件应该搜索lib-common。最后一个include指明包含的头文件的依赖关系（nemu中头文件包含关系确实比较杂）
define make_common_rules
$(1)_SRC_DIR := $(1)/src
$(1)_INC_DIR := $(1)/include
$(1)_OBJ_DIR := obj/$(1)

$(1)_CFILES := $$(shell find $$($(1)_SRC_DIR) -name "*.c")
$(1)_SFILES := $$(shell find $$($(1)_SRC_DIR) -name "*.S")

$(1)_COBJS := $$(patsubst
$$($(1)_SRC_DIR)%.c,$$($(1)_OBJ_DIR)%.o,$$($(1)_CFILES))
$(1)_SOBJS := $$(patsubst
$$($(1)_SRC_DIR)%.S,$$($(1)_OBJ_DIR)%.o,$$($(1)_SFILES))
$(1)_OBJS := $$($(1)_SOBJS) $$($(1)_COBJS)

$(1)_BIN := $$($(1)_OBJ_DIR)/$(1)

$(1)_CFLAGS = $(CFLAGS) -I$$($(1)_INC_DIR) $(2)
$(1)_ASFLAGS = -m32 -MMD -c -I$$($(1)_INC_DIR) -I$(LIB_COMMON_DIR)

$$($(1)_OBJ_DIR)%.o: $$($(1)_SRC_DIR)%.c
        $$(call make_command, $(CC), $$($(1)_CFLAGS), cc $$<, $$<)

$$($(1)_OBJ_DIR)%.o: $$($(1)_SRC_DIR)%.S
        $$(call make_command, $(CC), $$($(1)_ASFLAGS), as $$<, $$<)

-include $$($(1)_OBJS:.o=.d)
endef
```

```
#这里调用了make_common_rules，已经生成了需要的依赖关系，这个文件只需要写明如何进行链接就行了
#最后说明-E的编译选项，用于展开宏定义便于代码的阅读和理解。
nemu_CFLAGS_EXTRA := -ggdb3 -I$(LIB_COMMON_DIR)
$(eval $(call make_common_rules,nemu,$(nemu_CFLAGS_EXTRA)))

nemu_LDFLAGS := -lreadline -lSDL

$(nemu_BIN): $(nemu_OBJS)
        $(call make_command, $(CC), $(nemu_LDFLAGS), ld $@, $^)
#       $(call git_commit, "compile NEMU")


##### rules for generating some preprocessing results #####

PP_FILES := $(filter nemu/src/cpu/decode/%.c nemu/src/cpu/exec/%.c,
$(nemu_CFILES))
PP_TARGET := $(PP_FILES:.c=.i)

.PHONY: $(PP_TARGET)

$(PP_TARGET): %.i: %.c
        $(call make_command, $(CC), -E -I$(nemu_INC_DIR), cc -E $<, $<)

cpp: $(PP_TARGET)

clean-cpp:
        -rm -f $(PP_TARGET) 2> /dev/null
```

IA32指令解析例子

在读取到一条指令时如何知道这条指令哪里开始，哪里结束，最后表达了什么意思，这就是这一节讨论的东西。我们将要把二进制反汇编为汇编语言，即通过CPU指令的构造原理将指令的二进制代码转化为助记符。

考虑这样一个代码片段，`eb 00 eb fe 90`

```
1c8d:0100 eb00      jmp short 0102
1c8d:0102 ebfe      jmp short 0102
1c80:0103 90        NOP
```

eb是jmp short指令的opcode，之后1个字节是相对跳转偏移，由于每次执行一条指令，cpu会自加上指令长度，这里的jmp short为两个字节，根据相对跳转的规则`0100 + 2 + 0 = 0102` 所以最后执行指令的地址为。第二条指令也是jmp short指令，而相对跳转偏移0xfe，所以最后`0fe + 2 = 0x100` 由于是short取低两位，所以最后的地址0102该地址本身。最后一条指令是一条NOP，什么也不做。

进一步探究指令操作码的构造：

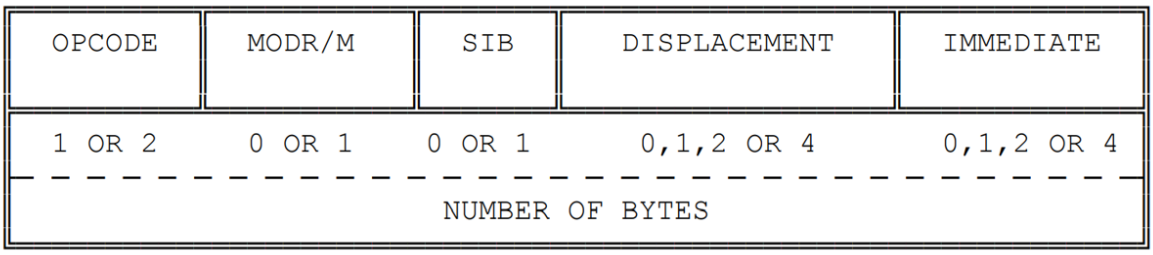
```
b80000      mov ax, 0
b80100      mov ax, 1
bb0000      mov bx, 0
bb0100      mov bx, 1
```

这里可以看见b8和bb分别代表了`mov ax, num`和`mov bx,num`,寄存器的信息已经包含在操作码内了。

```
66b800000000        mov eax,00  #这是80386的16位模式下的代码
```

但是在16位机器上，将会解释为

```
66      db 66                   #如果假设在16位机上debugger会将66独立翻译
b80000  mov ax, 0
0000    mov [bx+si], al
```



在x86架构上，一条指令可以由6个部分构成：

1. Prefixes 指令代码前缀，每指令最多有可以有4个前缀修饰。有操作数大小前缀，如前面提到的0x66, 在16位模式下，它将指定32-bit的操作数。有重复型前缀，如常见的F3表示REP重复前缀，F2便是PRENZ前缀。还有段超越（覆盖或者重写更符合意思），2E对应CS、36对应SS、3E对应DS、26对应ES、64和65分别对应FS、GS，段超越可以指定内存寻址时所用的段寄存器。还有寻址地址大小前缀，0x67。
2. opcode本体，它可以是一个字节，也可以0x0f扩展，或者在ModR/M中实现扩展。
3. ModR/M 主要用于指定寻址模式，x86支持直接寻址、寄存器寻址、变址寻址等等，这些寻址方式就是由这个域还有另外一个SIB指定的。高2bits位mod位，中间3bits位Reg/Opcode 位，最低3bits位R/M位。
4. SIB，和ModR/M一起指定寻址模式，SIB是Scale Index Base的意思，是作为寻址模式的补充，它被细分为三个域，高2bits的比例因子，中间3bits的Index索引用来指定变址寄存器，最低3bits的base寄存器用于指定基址。
5. Displacement偏移位，可以有1、2、4、8字节。
6. Immediate 立即数，可以有1、2、4、8字节。

接下来可以看一下add指令，指令如下所示：

```
Opcode    Instruction         Clocks    Description

04 ib     ADD AL,imm8          2        Add immediate byte to AL
05 iw     ADD AX,imm16         2        Add immediate word to AX
05 id     ADD EAX,imm32        2        Add immediate dword to EAX
80 /0 ib  ADD r/m8,imm8        2/7      Add immediate byte to r/m byte
81 /0 iw  ADD r/m16,imm16      2/7      Add immediate word to r/m word
81 /0 id  ADD r/m32,imm32      2/7      Add immediate dword to r/m dword
00 /r     ADD r/m8,r8          2/7      Add byte register to r/m byte
......
```

- 第一列指明了指令码的取值，可以看到同一条指令具有不同的表达形式。`/digit` 里的digit指的是0-7的数值，占用了ModR/M的reg域，所以此时ModR/M只使用了。`/r` 表明指令的寻址模式中指定了寄存器间接寻址和R/M操作数。`ib`、`iw`、`id` 分别表示立即数为 1-Byte、2-Byte、4-Byte。
- 第二列指令了指令的汇编代码的解释，下面是对操作数的解释：
  - rel8： 表示操作数是一个8-bit 相对寻址，范围在相同的代码段，当前指令末端的前128-Byte到 127-Byte。 rel16和rel32作用和rel8类似，只是操作数的大小不同，范围也更加大了。
  - ptr16:16, ptr16:32 ：表示了远代码指针，操作数包含了代码段地址和偏移，用于代码寻址，CPU会按照指针的代码段地址来设置CS寄存器来实现远跳转。
  - r8：表示一个 8-bit 的通用寄存器 AL, CL, DL, BL , AH, CH, DH, BH。
  - r16：表示一个 16-bit 通用寄存器 AX, CX, DX, BX, SP, BP, SI, DI。

    - r32：表示一个 32-bit 通用寄存器 EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI。
    - imm8：表示一个 8-bit 立即数， imm16和imm32也类似。
    - r/m8: 表示操作数可以为8-bit的内存数据或者寄存器。r/m16和r/m32也同理。
    - m8: 表示8bit的内存数据，由DS:SI或者ES:DI寻址得到，专门用于字符串指令。
    - m16:16，m16:32 : 表示操作数为远指针寻址，冒号前的值为段基址，后面的值为相应偏移。
    - m16 & 32, m16 & 16，m32 & 32: 指明了左右操作数的到校，如m16 & 32 就被LIDT 和 LGDT使用。
    - moffs8, moffs16, moffs32: 表示一个内存偏移值， 对于不含 ModR/M 的指令，将在指令码中包含内存偏移信息，它同时决定了地址寻址方式。
    - Sreg： 表示了一个段寄存器， ES=0, CS=1, SS=2, DS=3, FS=4, GS=5。
- 第三列、第四列分别说明了指令所消耗的时钟周期和指令的功能。文档中还包含了其他详细的内容，如Operation使用伪代码描述了CPU内部执行指令的过程，Flags Affected描述了标志寄存器受指令影响的标志位，以及指令是否触发异常的信息（Nemu中不实现）等等。

以 dec 指令为例， dec的指令码 48 或者 FE/1后者FF/1，以48为opcode是典型的寄存器嵌入opcode，这种指令的操作数一定是寄存器。FE/1 或者 FF/1是是典型的 reg扩展指令，在ModR/M的reg用来扩展opcode，这个类型的opcode可以通过R/M和mod域指明寻址模式，操作数的内存可以内存。

```
1C8D:0100 48      dec ax
1C8D:0101 FEC8    dec al
1C8D:0103 4B      dec bx
1C8D:0104 FECB    dec bl
1C8D:0106 49      dec cx
1C8D:0107 FEC9    dec cl
```

又以汇编代码为例：

```
1C8D:0100 67668B00    mov eax,[eax]
1C8D:0104 678B00      mov ax, [eax]
1C8D:0107 678A00      mov al, [eax]
1C8D:010A 67668900    mov [eax], eax
1C8D:010E 678900      mov [eax], ax
1C8D:0111 678800      mov [eax], al
1C8D:0114 66A3        mov eax, eax
```

先来分解第一条指令（不知为什么找的例子中，它要以16位模式为例子），首先，看第一个字节0x67指明了寻址位数，16为模式下，这意味着要使用32位寄存器寻址。再看0x66是一个指令前缀，指明了操作数位32-bit。再次根据手册定义8B是mov的指令码，接下来的0x00就是ModR/M位，得到Mod = 0，Reg/opcode = 0，R/M = 0 。对应到32-bit的寻址模式表中，找到Mod=0行，在Mod=0的子表格，在根据RM指定的行和reg域指定的列最终确定了 [eax] 和 eax。

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 Base = Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

再次实践上述表格的内容：

- `mov eax, [edx]`

汇编上面的指令，仍然考虑在16位模式下，这条指令使用了32位寻址还使用了32位操作数，根据手册找到`8B /r  mov r32,r/m32`这条指令，根据[EDX]和eax去ModR/M表格中去寻找，得到reg = 0，ModR/M = 02, 于是我们得到汇编的结果`67 66 8b 02`。

- `mov [eax], edx`

  再次查表可得opcode 应该为`0x89`，根据 [eax] 和 edx 去查表，我们可以先确定寻址模式位mod = 0，再根据操作数具体确定行、列坐标，最终我们得到了汇编结果`67 66 89 10`。接下来尝试没有内存寻址的指令。

- ```
  mov edx, eax   // 668BD0
  mov eax, edx   // 668BC2
  ```

  我们来考察到该使用`mov_rm2r`还是`mov r2rm`，在使用`mov`把一个寄存器的值转移到另外一个时，考察汇编的结果可以确定使用的是`mov_rm2r`。具体来看，`0xd0`的二进制表示为`0b11010000`,使用的mod = 3，得到使用的是寄存器寻址，并且源寄存器为0号寄存器 eax，目的寄存器为2号寄存器 edx。

- ```
  Opcode    Instruction       Clocks   Description
  F6 /6     DIV AL,r/m8       14/17    Unsigned divide AX by r/m byte(AL=Quo, AH=Rem)
  F7 /6     DIV AX,r/m16      22/25    Unsigned divide DX:AX by r/m word (AX=Quo, DX=Rem)
  F7 /6     DIV EAX,r/m32     38/41    Unsigned divide EDX:EAX by r/m dword (EAX=Quo, EDX=Rem)
  ```

  考察div指令，div通过ModR/M的reg域来扩展 opcode，而由于 div 默认使用eax寄存器所以 reg 用来扩展 opcode 不会有什么影响，只需在 R/M 说明显示使用的寄存器或者内存地址就可以了。

最后进行一次手工的反汇编来结束IA32指令的例子解析：

```
00000000 EB 5E 80 00 20 39 FF FF 00 00 00 00 00 00 00 00 .^.. 9..........
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
~ Duplicate Lines
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000060 FA 31 DB 8E D3 BC 80 05 E8 00 00 5B 81 EB 6B 00 .1.........[..k.
00000070 C1 EB 04 8C C8 01 C3 8E DB 53 6A 7D CB 68 00 20 .........Sj}.h.
```

cpu 执行代码时首先读入了 `eb`，可以得知这是一条跳转指令，且操作数为一个字节，随意 `5e` 就是偏移，由于指令本身的长度为2，所以最终跳转的地址为 `0x60` 。在这个地址处，读入一个字节 `fa` ，则会使一条cli指令，关闭了中断。接着读入 `31` ，指令的 xor r2rm，而ModR/M域的内容为 `db` 对应的二进制表达为 `0b11011011` ，而且mod域为3，reg域为3，R/M域为3，所以指令 `xor bx,bx` (指令在16位模式上运行)。紧接着一个字节 `8e` 代表了 `mov sreg, r/m16` ，ModR/M为 `d3` 对应的二进制为 `0b11010011` 所以mod为3，reg为2，RM为3，最终的指令就是 `mov ss, bx` 。紧接着就是 `bc` ,这条指令时典型的操作数嵌入opcode，即opcode中已经指明了一个操作数，这种指令要么是从模16余0开始连续八个，要么就是模16余8开始连续的8个，在此处我们将 `bc - b8` 得到4，推导出寄存器位 SP,由于在16位模式下，可以接下来的2bytes是另一个操作数，最终的得到指令 `mov sp,0x580` 。然后一个字节是 `e8` ，代表了一条 `call` 指令，由于是在16位模式下，所以整条指令占用3个字节， 读入两个字节为 `0` ，所以实际的跳转地址为下一条指令。下一条指令的opcode为 `5b` ，又是一条操作数嵌入opcode的指令，由 `5b - 58` 得到 3 ，所以最终的指令为 `pop bx` 。然后一条指令的opcode是 `81` ,这条指令需要通过ModR/M进行扩展，我们再读入一个字节，可以得到ModR/M的reg域为5，所以因该是sub指令，最终16位模式上得到的指令就是 `sub bx, 0x6b` 。

Nemu中如何实现指令解析

Prefixes 指令代码前缀，每个指令最多可以由4个前缀。

```
--------------+------------+------------+---------+--------+-------+-------+-----
----------+-------------+
| instruction | address-   |   operand- | segment | opcode |ModR/M |  SIB  |
displacement  |    immediate    |             |            |            |
  |        |       |       |               |
|    prefix   |size prefix | size prefix| override|        |       |       |
          |                |             |            |            |         |
  |       |       |               |
--------------+------------+------------+---------+--------+-------+-------+-----
----------+--------------+
|   0  or  1  |   0  or 1  |  0  or  1  |  0 or 1 | 1 or 2 | 0 or 1| 0 or 1| 0 or
1 or 2   | 0 or 1 or 2    |             |            |            |         |
  |       |       |    or 4       |    or 4      |
|-------------------------------------------------------------------------------
-------------------------|
```

Nemu中实现了instruction prefix，operand-size prefix。
这里先给出 ModR/M和SIB的结构:

```
ModR/M
7   6    5  4  3    2  1  0
+-----+----------+---------+
| mod | reg/opcode|   r/m   |
+-----+----------+---------+

SIB
+-----+----------+---------+
|  ss |   index  |   base  |
+-----+----------+---------+
```

ModR/M和SIB用于说明寻址模式。

从opcode说起

```
make_helper(exec) 进行宏展开获得 int exec(swaddr_t eip)，函数体为
int exec(swaddr_t eip)
{
  ops_decoded.opcode = instr_fetch(eip, 1);  //从dram里读取一个字节
    return  opcode_table[ ops_decoded.opcode ](eip); //opcode_table是一个函数地址组成的表
 }
 //于是通过opcode表就可以实现对opcode的解析，返回值在cpu_exec中可以对cpu.eip加上指令长，于是eip指向下一条指令。
 //但是i386中还存在指令扩展和reg域的扩展。

 static int _2byte_esc(swaddr_t eip) {
      eip ++;
      uint32_t opcode = instr_fetch(eip, 1);
      ops_decoded.opcode = opcode | 0x100;
      return _2byte_opcode_table[opcode](eip) + 1;
}
//在opcode中有一个特殊的函数，即0x0f中的函数_2byte_esc执行这个函数会再读取一个字节，将这个字节作为新的opcode在_2byte_opcode_table
//中查找执行函数。

#define make_group(name, item0, item1, item2, item3, item4, item5, item6, item7) \
    static helper_fun concat(opcode_table_, name) [8] = { \
    /* 0x00 */  item0, item1, item2, item3, \
    /* 0x04 */  item4, item5, item6, item7  \
    }; \
    static make_helper(name) { \
        ModR_M m; \
        m.val = instr_fetch(eip + 1, 1); \
        return concat(opcode_table_, name) [m.opcode](eip); \
    }
//这个宏用于定义opcode和后面ModR/M中的reg共同确定指令的情况，可以看到reg有三位，那么一个group就可以对应8个函数，生成一个8个函数的opcode表，
通过再读取一个字节获取reg就可以在8个函数中选择一个进行执行。
```

指令的解析

在exec中执行后，首先在all-instruction.h中包含了所有指令的头文件。

```
//header

这里以mov为例，在mov.h中声明了所有在opcode中声明了mov指令所有可以能遇到的情况，
i表示立即数，r表示寄存器，rm表示寄存器或者存储，2表示to，b表示单个字节，v表示16
字节或者32字节。
make_helper(mov_i2r_b);
make_helper(mov_i2rm_b);
make_helper(mov_r2rm_b);
make_helper(mov_rm2r_b);
make_helper(mov_a2moffs_b);
make_helper(mov_moffs2a_b);

make_helper(mov_i2r_v);
make_helper(mov_i2rm_v);
make_helper(mov_r2rm_v);
make_helper(mov_rm2r_v);
make_helper(mov_a2moffs_v);
make_helper(mov_moffs2a_v);
```

```
// source

以jmp的.c文件为例子，最下方的make_helper_v用于对16还是32位进行解析，依据就是指令operand是否
有
operand-size prefix即0x66，由于nemu默认在32位下运行如果没有0x66改变操作数大小，这里将选择
执行
32位如jmp_rm_l执行，否则执行jmp_rm_w。
由于同一类的函数的执行过程相同，而很多指令的操作数解码操作相同，这里使用模板进行归纳，三次模板生
成b,w,l
三种指令，值得注意有的指令没有b类指令，有的指令的两个操作数并不是相同大小的。
#include "cpu/exec/helper.h"

#define DATA_BYTE 1
#include "jmp-template.h"
#undef DATA_BYTE

#define DATA_BYTE 2
#include "jmp-template.h"
#undef DATA_BYTE

#define DATA_BYTE 4
#include "jmp-template.h"
#undef DATA_BYTE

make_helper_v(jmp)
make_helper_v(jmp_rm)
```

```
// template

这里依旧以mov的模板文件为例，模板的头文件根据DATA_BYTE生成一些模板所需要的宏，
当让结束时，也要取消这些宏，因为这些宏是根据DATA_BYTE产生的，DATA_BYTE被undef了，那么
这些宏也要被undef
#include "cpu/exec/template-start.h"
```

```
#define instr mov

static void do_execute() {
    OPERAND_W(op_dest, op_src->val);//这是一个宏，将值写到目标寄存器或者地址
    print_asm_template2();
}

//make_instr_helper用于decode之后执行，之所以要传入i2r，i2rm是因为要根据操作数类型进行
decode函数的执行，
//而统一类指令的执行大多类似，很多可以用一个do_execute概括。

make_instr_helper(i2r)
make_instr_helper(i2rm)
make_instr_helper(r2rm)
make_instr_helper(rm2r)

//特殊的可以单独写，这里主要是a2moffs没有单独的解码函数。
make_helper(concat(mov_a2moffs_, SUFFIX)) {
    swaddr_t addr = instr_fetch(eip + 1, 4);
    swaddr_write(addr, DATA_BYTE, (cpu.gpr[check_reg_index(R_EAX)]._32));

    print_asm("mov" str(SUFFIX) " %%%s,0x%x", REG_NAME(R_EAX), addr);
    return 5;
}

make_helper(concat(mov_moffs2a_, SUFFIX)) {
    swaddr_t addr = instr_fetch(eip + 1, 4);
    REG(R_EAX) = MEM_R(addr);

    print_asm("mov" str(SUFFIX) " 0x%x,%%%s", addr, REG_NAME(R_EAX));
    return 5;
}

#include "cpu/exec/template-end.h"
```

解码细则

这里可以归为到上一个标题，但是过于重要，单独归为一部分。x86指令结构的解析，主要就是这一部分。

```
先说明ModR/M和SIB的解析，这两个域的解析由read_ModR_M和load_addr实现

int read_ModR_M(swaddr_t eip, Operand *rm, Operand *reg) {
    ModR_M m;
    m.val = instr_fetch(eip, 1); //读取modR/M
    reg->type = OP_TYPE_REG;     //可以确定一个操作数，如果有使用到一定是寄存器
    reg->reg = m.reg;

    if(m.mod == 3) {          //mod可有4个值，mod为3时，就是两个寄存器将的操作
        rm->type = OP_TYPE_REG;
        rm->reg = m.R_M;
        switch(rm->size) {
            case 1: rm->val = reg_b(m.R_M); break;
            case 2: rm->val = reg_w(m.R_M); break;
            case 4: rm->val = reg_l(m.R_M); break;
            default: assert(0);
```

```
        }
#ifdef DEBUG                            //打印到assembly中用于debug
        switch(rm->size) {
            case 1: sprintf(rm->str, "%%%s", regsb[m.R_M]); break;
            case 2: sprintf(rm->str, "%%%s", regsw[m.R_M]); break;
            case 4: sprintf(rm->str, "%%%s", regsl[m.R_M]); break;
        }
#endif
        return 1;
    }
    else {
        //如果不是两个寄存器将的操作那么就涉及访存了
        int instr_len = load_addr(eip, &m, rm);
        rm->val = swaddr_read(rm->addr, rm->size);
        return instr_len;
    }
}

//RM为4，那么就需要读取SIB
//这里扩展的原理或则说原则就是EBP不可作为base，ESP不可以做为index
int load_addr(swaddr_t eip, ModR_M *m, Operand *rm) {//这里的eip是取完opcode后的地址
    assert(m->mod != 3);

    int32_t disp;
    int32_t instr_len, disp_offset, disp_size = 4;
    int base_reg = -1, index_reg = -1, scale = 0;
    swaddr_t addr = 0;

    if(m->R_M == R_ESP) {
        SIB s;
        s.val = instr_fetch(eip + 1, 1);
        base_reg = s.base;
        disp_offset = 2;
        scale = s.ss;

        if(s.index != R_ESP) { index_reg = s.index; }
    }
    else {
        /* no SIB */
        base_reg = m->R_M;
        disp_offset = 1;    //disp_offset为1或者2，即opcode到disp有几个字节
    }

    if(m->mod == 0) {
        //mod为零，base为5留出来用于直接寻址[DISP]
        if(base_reg == R_EBP) { base_reg = -1; }//直接寻址
        else { disp_size = 0; }
    }
    else if(m->mod == 1) { disp_size = 1; }

    //取偏移
    instr_len = disp_offset;
    if(disp_size != 0) {
        /* has disp */
        disp = instr_fetch(eip + disp_offset, disp_size);
        if(disp_size == 1) { disp = (int8_t)disp; }

        instr_len += disp_size;
```

```
        addr += disp;
    }

    if(base_reg != -1) {
        addr += reg_l(base_reg); //基地址
    }

    if(index_reg != -1) {
        addr += reg_l(index_reg) << scale; //变址
    }

#ifdef DEBUG
    char disp_buf[16];
    char base_buf[8];
    char index_buf[8];

    if(disp_size != 0) {
        /* has disp */
        sprintf(disp_buf, "%s%#x", (disp < 0 ? "-" : ""), (disp < 0 ? -disp :
disp));
    }
    else { disp_buf[0] = '\0'; }

    if(base_reg == -1) { base_buf[0] = '\0'; }
    else {
        sprintf(base_buf, "%%%s", regsl[base_reg]);
    }

    if(index_reg == -1) { index_buf[0] = '\0'; }
    else {
        sprintf(index_buf, ",%%%s,%d", regsl[index_reg], 1 << scale);
    }

    if(base_reg == -1 && index_reg == -1) {
        sprintf(rm->str, "%s", disp_buf);
    }
    else {
        sprintf(rm->str, "%s(%s%s)", disp_buf, base_buf, index_buf);
    }
#endif

    rm->type = OP_TYPE_MEM;
    rm->addr = addr;


    //addr = base + index << scale + displacement,其中没有SIB时，mod == 0时，分为R_M
为5和不为5，为5的话直接[DISP32],不为5的话[REG(R_M)]，
    //如果mod不为0，则都有disp，mod为1时是8位，mod为2是32位，bas都是REG(R_M)，在读取完
DISP后，生成[REG(R_M)+DISP],否则在RM为4时则解析出base
    //index,scale形成[Base + index * ss + disp]
    return instr_len;
}
```

最后在decode函数中遇到要解析地址的时候，调用read_ModR_M()。

指令的解释执行

还是以jmp为例

```
static void do_execute(){
    if(ops_decoded.opcode==0xff)  //不同的jmp操作不同
    {
        int len = concat(decode_rm_,SUFFIX)(cpu.eip+1)+1;  //decode解码，获取指令长度
        cpu.eip = (uint32_t)(DATA_TYPE_S)op_src->val - len ;//跳转，将eip重新赋值，-len是因为每个指令循环后eip会自动加上指令长度
    }
    else
        cpu.eip = cpu.eip + (uint32_t)(DATA_TYPE_S)op_src->val;//相对跳转
    print_asm_template1();
}

//注意地址跳转时，做运算记得使用符号扩展
```

指令实现中的易错点

```
//string中的rep，cmpsb和cmpsw可能提前退出
if((ops_decoded.opcode == 0xa6 || ops_decoded.opcode = 0xa7))
    break;
```

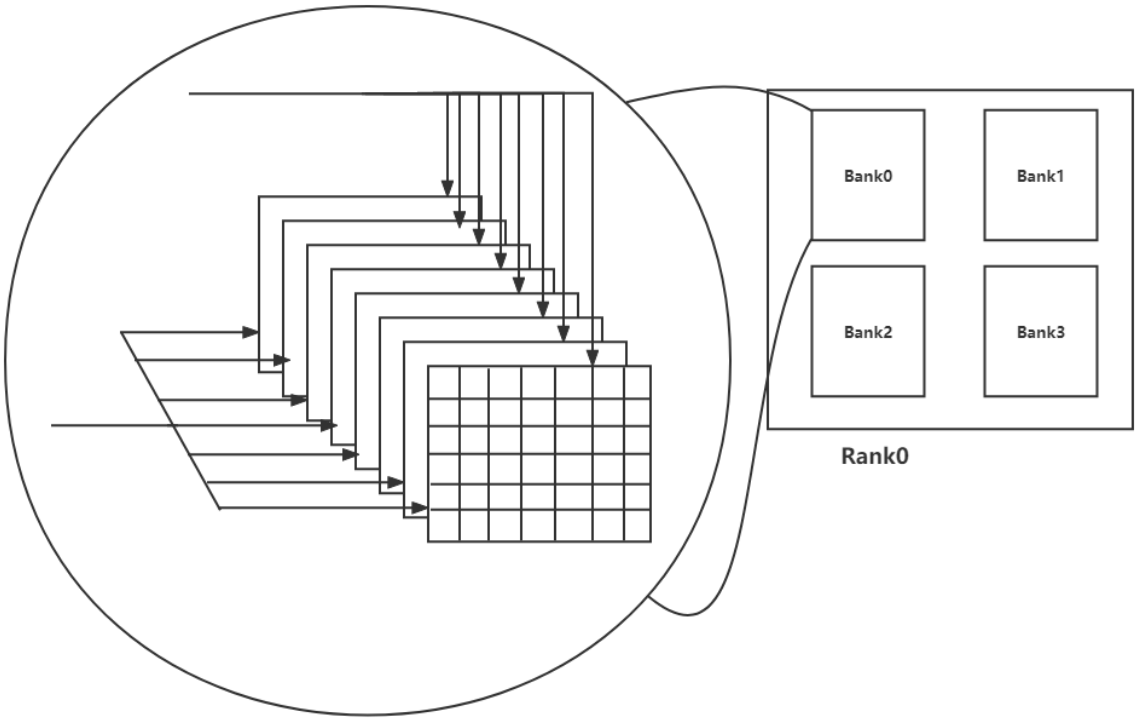
```
//arith中inc和dec的溢出判断是不同的，如inc的溢出判断只用这么写
op_src->val == (((DATA_TYPE)0x01)<<(8*DATA_BYTE-1)) - 1
//实现cmps时，记住更新eflags
//sbb中是减去eflags.CF用于借位
DATA_TYPE result = op_dest->val - op_src->val - eflags.CF;
//实现aad指令时，低位直接&0xff，结果不会影响ah
cpu.eax = (cpu.eax & 0xffffff00) + ((ah * val + ( cpu.eax & 0xff ))&0xff);
```

```
//control
//call之中，nemu的eip自加是在执行一条指令之后的，于是直接跳转的地址而不是相对跳转的清况，要记得减去指令长度，否则会出问题，同时call有两种类型一个是相对当前地址的跳转，一个是直接跳转到给定地址，必须做一个判断。
//hlt中要记得检查键盘消息
//int 指令需要直接改变eip，因为中断跳转之后不会进行eip自加了，同时还有一个小的点，如果想要打印这条指令的地址，必须在指令函数的内部执行，否则debug时可能看不见信息。
//iret记得改变不可见部分
//push的时候记得没有push一个字节的情况，根据现在的模式选择16还32位。
```

```
//mov-data中
//lgdt记得要用线性地址
//mov register,cr0中间三位便是crX寄存器
//movsx的情况要在所有有符号立即数的情况也实现一遍
//stos指令的esi是不变的
```

PA3 存储管理

DRAM结构



NEMU中对ddr3进行了模拟,地址一共27位，rank地址4位，bank地址3位，行地址10位，列地址10位，那么内存的空间将一共有128MB的内存。

DRAM的内部，在每一个bank设定了一个行缓冲，每一个行缓冲寄存器，有一个row_id标记当前缓冲器内的行号，还有一个valid用于标记当前缓冲是否被使用。

```
typedef struct{
  uint8_t buf[NR_COL];
  int32_t row_idx;
  bool valid;
} RB;

一个类的存储用的是uint8_t说明一个bank中有8个存储矩阵，一个列存储8个bit。
```

在ddr3开始要确保每个换缓冲中的valid位为false，即没有被使用。

Dram的读

dram的读是一个burst的读，即猝发传送，读地址连续的burst_len个字节，在nemu的模拟中burst_len为8，然而burst的起始地址并不是任意的，它必须是8的整数倍，所以一个burst是读这个区间上的8个字节。

代码如下：

```
  if(!(rowbufs[rank][bank].valid&&rowbufs[rank][bank].row_idx==row)){
    memcpy(rowsbuf[rank][bank].buf,dram[rank][bank]);
    rowbufs[rank][bank].row_idx = row;
    rowbbufs[rank][banl].valid = true;
  }
  //这里的列地址的第三位为0
  memcpy(data,rowbufs[rank][bank].buf + col,BURST_LEN);
```

dram_read函数要在这个基础上做进一步的抽象，因为dram_read的参数一个是地址，一个读取的长度，长度是个不确定的值。这个地址不一定是8的整数倍，所以可能要分多次读取，在nemu中长度进行限制，无论如何最多四个字节，最多分两个burst读取。

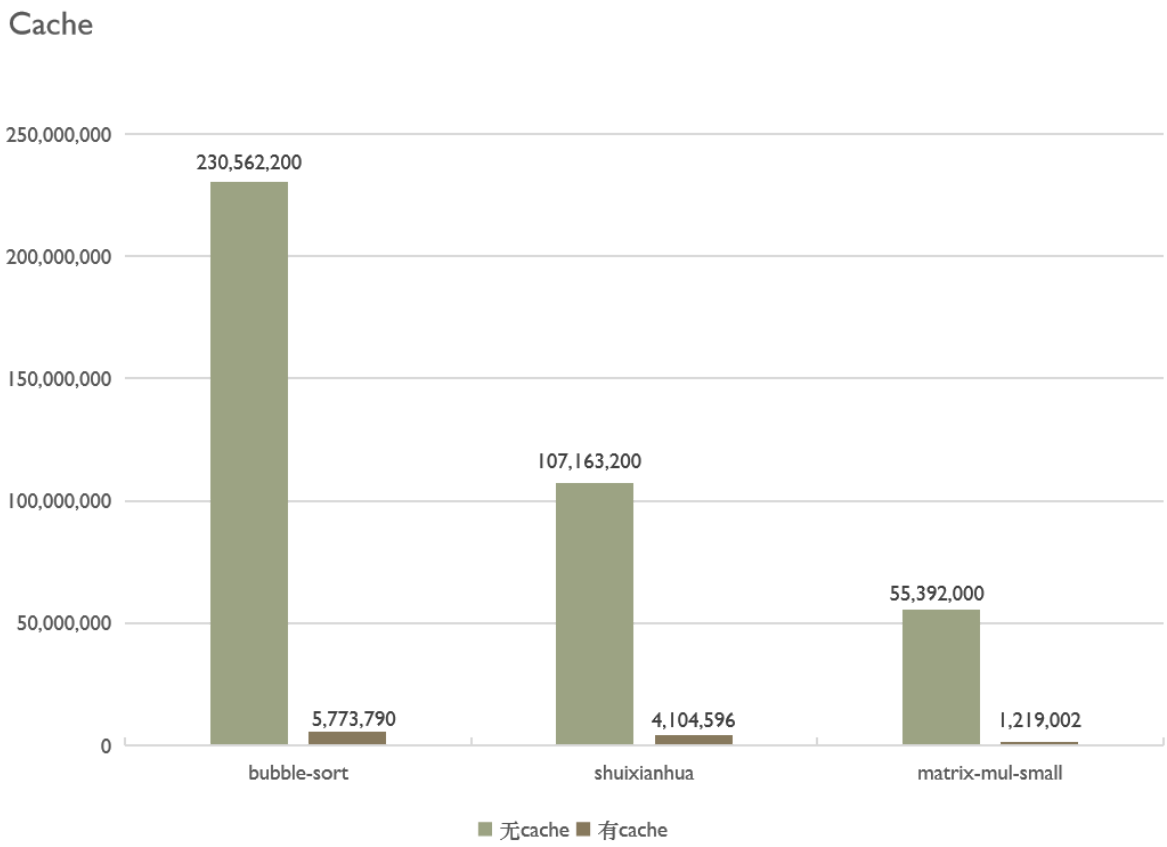
Dram的写

dram的写相比于读的特殊性在于不经要把写的行读入缓冲进行修改，还要写入bank，最后由于burst的内容不能全写，需要一个掩码mask，指出一个burst长度中需要写的部分。

Cache

Cache是计算机组成原理的学习中的重点，只是简单介绍本实验中实现的Cache用到的内容。Cache的机制利用了局部性原理：第一部分空间局部性，程序在访问一个地址后，接下来很可能访问周围的地址；第二部分时间局部性，程序在访问一个地址后，接下来很可能再次访问。这个用来说明局部性原理的例子往往是一个循环数组访问，循环变量 `i` 每次循环都需要访问，同时一个数组总是被一个接一个的访问，在物理空间上这些元素往往相邻存储，这一个例子中很好体现了局部性原理。

在Nemu中实现了两级Cache，即L1和L2。L1 Cache 采用了随机替换、组相联映射、直写法和非写分配；L2 Cache采用了随机替换、组相联映射、写回法、写分配。这些算法的采用都是权衡，最后经过只采用L1 cache，可以在Nemu中实现 98% 以上的命中率，执行的时钟周期至少可以到原来二十分之一。



算法	命中率
bubble-sort	99.89%
shuixianhua	99.81%
matrix-mul-small	98.81%

C与面向对象

C本身并不支持面向对象，但是C本身的灵活性允许我们自己实现面向对象。面向对象三大概念：封装、继承、多态。C中的访问隔离往往是通过文件确定的，这个在Nemu中十分明显，我们也并不实现这个封装的特性。而继承是可以选择实现的，C实现继承往往是通过proto原型实现的，proto是一个属性表，每一个结构体内都会一个proto原型，继承时只需要获取父类的proto到自己的proto就行了。而多态可以通过自己修改proto中函数就行了，proto中的函数都是函数指针，我们将函数指针附上不同的函数实现呢，那么就可以实现一个接口不同实现。

```
void (*init)(struct Cache *self);
uint32_t (*replace_algo)(struct Cache *self,uint32_t addr);
uint32_t (*map_algo)(struct Cache *self,swaddr_t addr);
uint8_t (*cache_read)(struct Cache *self,uint32_t bid,uint32_t offset);
uint32_t (*read)(struct Cache *self,uint32_t addr,uint32_t len);
void (*write_defect)(struct Cache *self,swaddr_t addr,void* data,uint32_t len);
void (*cache_write)(struct Cache *self,uint32_t bid,uint8_t data,uint32_t offset);
void (*write)(struct Cache *self,uint32_t addr,void* data,uint32_t len);
```

Nemu的实现中，我们加入8个函数，每个cache都要实现这8个函数：

- init：指明了函数的初始化操作，不同的cache可能不同。
- replace_algo: 用于在要替换一个cache时，确定要替换哪个cache，返回替换的block id号（在Cache中我们为所有的block 附上了一个id，尽管采用了组相联映射，block其实是一个一维数组）。

- map_algo: 跟据地址生成一个映射的block id号。
- cache_read: 用于直接读取一个Cache块。
- cache_write: 用于直接写一个Cache块。
- read：向上提供的抽象，为cpu提供读的服务，这里采用的物理地址，并未采用虚拟地址作为cache映射的依据。
- write：向上提供的抽象， 为cpu提供写的服务，用的是物理地址。
- write_defect: 在写缺失时该如何操作，在Nemu中具体有直写法和写回法。

较为简洁的实现

在NEMU中实现一个cache, 它的性质如下:

- cache block存储空间的大小为64B
- cache存储空间的大小为64KB
- 8-way set associative
- 标志位只需要valid bit即可
- 替换算法采用随机方式
- write through
- not write allocate

在NEMU中实现一个L2 cache, 它的性质如下:

- cache block存储空间的大小为64B
- cache存储空间的大小为4MB
- 16-way set associative
- 标志位包括valid bit和dirty bit
- 替换算法采用随机方式
- write back
- write allocate

从read_default开始，作为向上的抽象，它的输入是一个地址一个长度，还有由于采用了面向对象的写法，我们默认传入一个指向自己的指针。

1. 作为 Cache 存储，我们所有的操作默认先要进行一次地址的映射，我们这里采用的是组相联映射，映射到一个组后，我们从这个组的最小下标开始遍历找一个index返回，如果命中我们直接返回。
2. 对于读未命中的情况，又可以具体分为两种情况。如果是cache块中已经没有可以用的块了，映射算法会返回index = -1；如果cache仍然有可以分配的块，通过块中的valid位我们可以知道这也是

未命中的一种情况。

3. 对cache块已满的情况，我们通过Cache替换算法再次映射一次，由于是直写法，所以写回时不用考虑写回下一层存储。于是读命中的两种操作，接下的操作就一模一样了，写入L1 cache块，并读相应的地址。

但是Cache块中存在越界的问题，即一个数据的一部分在前一个Cache，另一部分在后一个Cache，这时就要考虑读两次的问题，加上考虑命不命中的问题，这样组合起来判断就会变得复杂，这里采用了一种递归的办法，简化了这种判断。

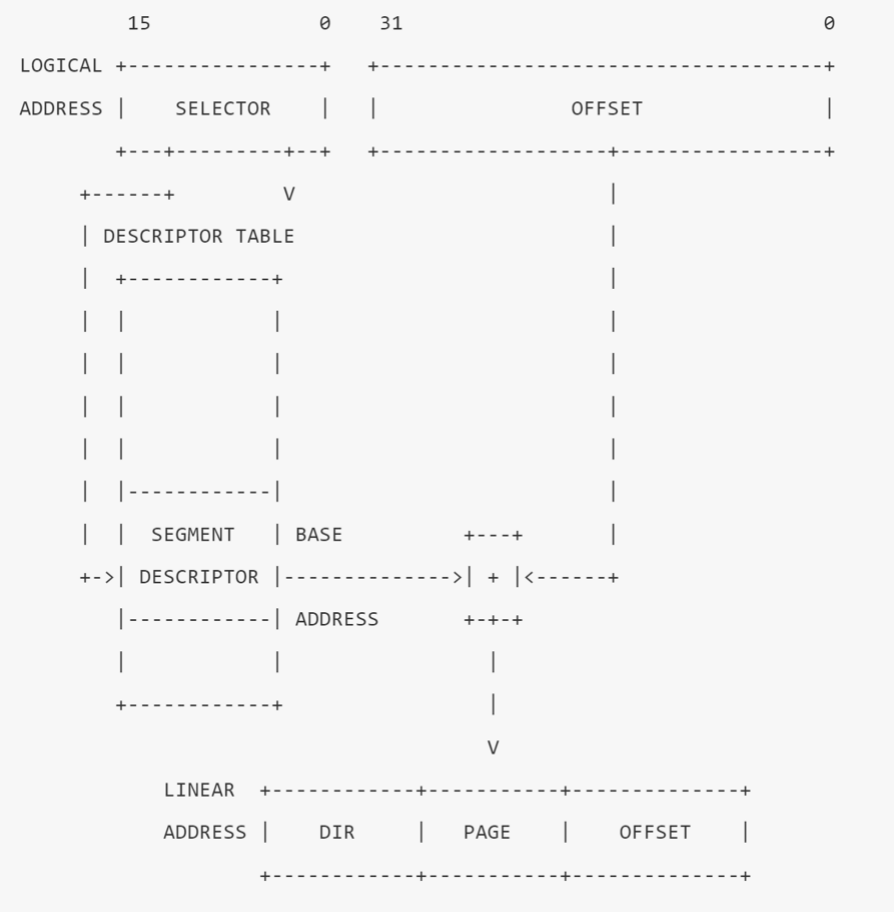
```
//根据上面所讲，每次读cache都是先来一次地址的映射，但是接下来我们直接上来判断是否越界，从而作为接下来是否递归的依据，实际上就是讲个一个读分为两次执行。我们发现即使两次读的格式完全一致，只是起始地址和长度不一样，意识我们将输入的len分为了(len1,len-len1)两个部分。
//第二合并的点在于，读未命中的时候，cache块满和不满，只有是否再映射一次的不同，而后面的做的操作是一致的，于是合并一次。
uint32_t read_default(Cache *self,uint32_t addr,uint32_t len){
    int32_t index = self->map_algo(self,addr),i,j,data;
    uint8_t tmp_data[4];
    bool cborder = addr%BLOCK_SIZE + len > BLOCK_SIZE? true : false;
    uint32_t len1 = cborder? BLOCK_SIZE - addr % BLOCK_SIZE : len;
    if(index == -1 || self->block[index].valid == false)//L1未命中
    {
        if(index == -1)
            index = self->replace_algo(self,addr);
        for(i=0;i<BLOCK_SIZE;i++)
        {
            uint8_t temp = cache_l2->read(cache_l2,addr/BLOCK_SIZE * BLOCK_SIZE
+ i,1);
            self->block[index].space[i] = temp;
        }
        self->block[index].valid = true;
        self->block[index].tag = addr/BLOCK_SIZE/((self-
>cache_size/BLOCK_SIZE)/self->set_way);
    }

    for(i=0; i< len1;i++)
        tmp_data[i]= self->cache_read(self,index,addr%BLOCK_SIZE + i);

    if(cborder) //越界了，递归再读一次，地址是下一个块的开头，长度为总长len - 已经读取的部分
    {
        data = read_default(self,addr+len1,len - len1);
        for(j=0;j<len-len1;j++,i++)
            tmp_data[i] = *((uint8_t*)&(data) + j);
    }

    return *((uint32_t*) tmp_data);
}
```

进入保护模式

对于保护模式主要是解决8086时代的两个问题：

- 8086是16机器，使用1MB的地址空间，我们要步入32位的世界。
- 8086中不同的段直接没有做权限隔离。

于是保护模式就来了，保护模式采用了GDT全局描述表的方式来解决以上的问题（LDT这里不做考虑）。即使80386仍然采用16的段寄存器，但是其中存的已经不是段基址了，我们称其中的部分位选择子Selector，当然为了实现更加快速和安全的访问，段寄存器有一个不可见的部分这也Nemu中要求实现的。开启了保护模式一个理想的寻址过程是这样的，每种操作往往有默认访问的寄存器，通过段寄存器中的选择子直接可以在GDT中找到一个全局的描述符，描述符中有真正的基址，在通过了权限和长度检查后，就可以实现地址的访问。由于这里已经是32位的时间，一个通用寄存器就可以有32位不需要再通过段寄存器左移4位加上偏移的方式来访存了，我们可以采用平坦模型将所有的段基址寄存器设为0，直接通过偏移就可以在4G的世界里遨游。现代操作系统里，可以直接将这些段直接分为内核数据段、内核代码段、用户数据段、用户代码段，全部运行在0基址上，段只有保持兼容和权限限制的功能。



不可见部分和权限

段寄存器可见的部分只有16位，但是不可见部分可以将整个段描述符加载进来，于是就可以减少一次访存，而不用每次都增加一次访存开销，加快了运行。但是段描述符中不只有段基地址，同时还有权限的信息，CPU中进行权限的检查时，并不用担心可见部分的第三位中的权限位被用户修改，每次CPU检查的都是不可见的权限 。

权限可以分位DPL、RPL、CPL 。CPL存在CS中的权限位，代表了当前进程的特权级；DPL表示了访问段的特权级，它位于GDT段描述中；RPL的不得不提到在遥远的时代，那时还不是在平坦模式下，当用户进程需要提供RPL告知系统自己要访问的存储空间在哪里，于是有一个恶意用户进程要启用系统调用，提供自己要访问的地址的选择子，陷入系统调用后，可以发现当前特权级为0，但是自己段寄存器中的选择子仍然是在3特权级，所以即使提供的参数在要访问0特权仍然通过RPL <= DPL的特权检查。

分段基址的实现

值得注意的是，在实验手册有这样一句话：

2. IA-32中规定不能使用 `mov` 指令设置CS寄存器, 但切换到保护模式之后, 下一条指令的取指就要用到CS寄存器了. 解决这个问题的一种方式是在 `restart()` 函数中对CS寄存器的描述符cache部分进行初始化, 将base初始化为0, limit初始化为 `0xffffffff` 即可.

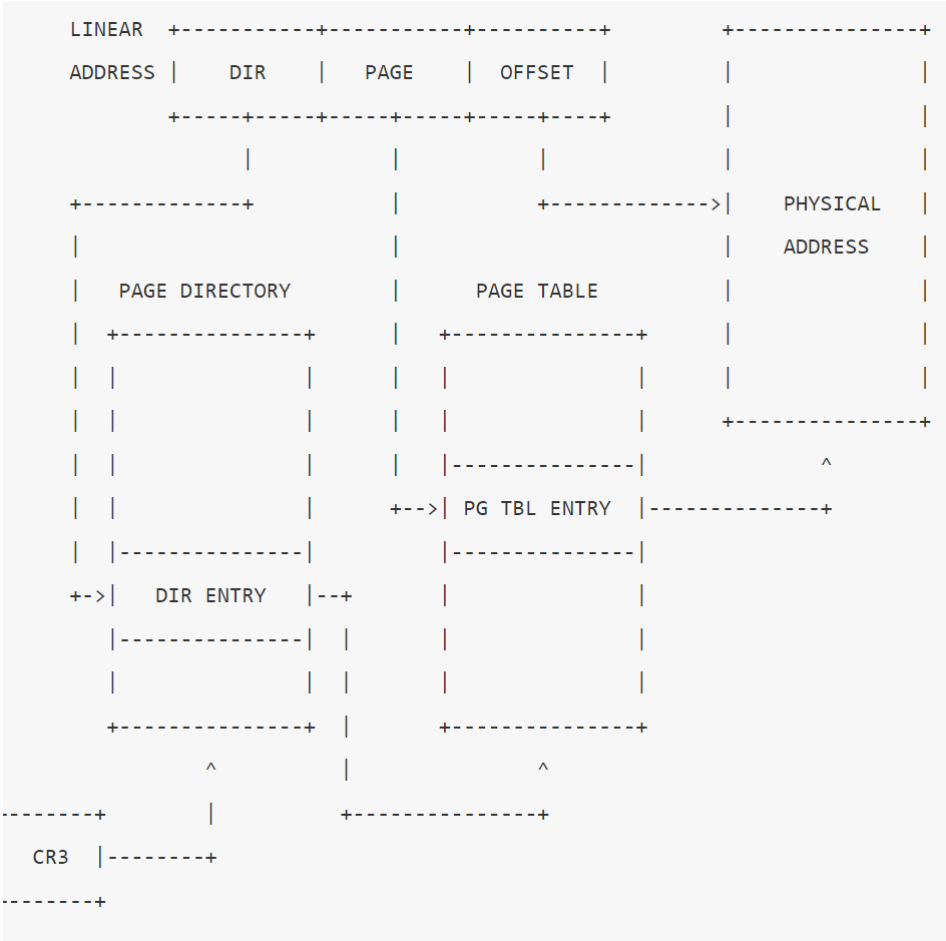
```
(nemu) si
  100000:   0f 01 15 48 00 10 00                lgdt 0x100048
(nemu) si
  100007:   0f 20 c0                            movl cr0,%eax
(nemu) si
  10000a:   83 c8 01                            orl $0x1,%eax
(nemu) si
  10000d:   0f 22 c0                            movl %eax,cr0
```

考虑实际执行的程序，在执行完 `movl %eax, cr0` 实际已经进入了保护模式，下一条指令的执行该如何确保段选择子不出错。Nemu使用了在一开始的restart中进行初始使得 cs 寄存器中的不可见部分被初始化成功，所以确保一进入保护模式就有正确的选择子。但是实际上的硬件可能是，流水线中已经有这条指令了，jmp不仅实现了cs加载还实现了流水线清空。

```
![page_translate](C:\Users\Porterlu\Desktop\picture\page_translate.png)//在实现了GDTR等寄存器和lgdt等指令后，nemu的访存单元只用在访存时加上这样一层转换就可以了，通过不可见部分的内容计算出基址即可。
lnaddr_t seg_translate(swaddr_t eip,uint32_t len, uint16_t sreg)
{
    uint32_t base,limit;![page_translate]
(C:\Users\Porterlu\Desktop\picture\page_translate.png)
    base = ((cpu.sr[sreg].gdesc>>32)&0xff000000) | ((cpu.sr[sreg].gdesc>>16)
&0x00ff0000) |\
                    ((cpu.sr[sreg].gdesc>>16)&0x0000ffff);

    limit = ((cpu.sr[sreg].gdesc>>32)&0x000f0000) | ((cpu.sr[sreg].gdesc &
0x0000ffff));
    if(cpu.sr[sreg].gdesc&0x0080000000000000)
        limit = (limit << 12) | 0x00000fff;
    assert((eip+len > eip) |( eip + len < limit))

    return base + eip;
}
```

最后记得在modrm.c中所有的地址访问加上段寄存器，执行指令时分为取指，取数据，访问栈都分别加上CS,DS,SS。

虚拟地址



在开启虚拟地址前，我们nemu的物理地址只有128MB，并不能充分使用32位的地址空间，但是通过两级页表的地址映射，我们终于可以把我们的内核（PA2我并没记录，将放在PA3中）移动到内核空间。通过CR3中的页目录地址和两次映射就可以的得到最后的物理地址，即线性地址最终被转化了物理地址。值得注意的是，页表上也有权限位，当一个在特权级3的进程试图访问一个特权级为0的页时也会发生错误。最终虚拟地址空间和物理地址空间不过就是就是两个集合之间的映射，page变换和seg变换看作一个复合函数的话，那么虚拟地址空间是定义域的话，物理地址空间就是值域。

$$hwaddr = page(seg(swaddr))$$

内核加载原理

我们的内核最终加载虚拟地址是已经确定的，它链接时设的虚拟地址必定在内核空间即虚拟地址的高位，但是我们的启动的第一条指令在0x100000。如何将一个这样一个程序实际运行在低地址就是第一个遇到的难题。已下就是Nemu的解决方案：

- Makefile.part 中的链接选项让kernel从虚拟地址 `0xc0100000` 开始
- NEMU 把kernel加载到物理地址 `0x100000` 处
- start.S 中 `va_to_pa()` 的宏让地址相关的部分减去KOFFSET

解析：在引入内核后，entry就已经是kernel了，我们将kernel放在模拟内存的0x100000处，指令中遇到需要将高地址转化为低地址的地方使用 va_to_pa()，减去相应的偏移，等到开启页表后，再跳转到高地址就可以正常运行，这里页表的构造是关键。2

kernel两次建立GDT，start.S 中程序一开始就通过减去KOFFSET方式运行在低地址。

```
.globl start
start:
    lgdt    (gdtdesc - KOFFSET) # See i386 manual for more
    movl    %cr0, %eax          # %CR0 |= PROTECT_ENABLE_BIT
    orl     $0x1, %eax
    movl    %eax, %cr0

# Complete transition to 32-bit protected mode by using lo
# to reload %CS and %EIP.  The segment descriptors are set
# translation, so that the mapping is still the identity m
    ljmp    $GDT_ENTRY(1), $va_to_pa(start_cond)

start_cond:
# Set up the protected-mode data segment registers
    movw    $GDT_ENTRY(2), %ax
    movw    %ax, %ds            # %DS = %AX
    movw    %ax, %es            # %ES = %AX
    movw    %ax, %ss            # %SS = %AX

# Set up a stack for C code.
    movl $0, %ebp
    movl $(128 << 20), %esp
    jmp init                # never return
```

可以看到我们将 `gdtdesc - KOFFSET` 作为了地址，分析原理，`gdtdesc` 在链接时被链接到了高地址，意味着 `gdtdesc` 是一个 `0xc01xxxxx` 的值，我们将这个减去 `KOFFSET` 的到了一个低地址的值，这个值为 `0x1xxxxx` 这个值就是我们加载到模拟内存的物理地址，究其根本就是因为一个object内部的相对位置是一定的，虚拟0xc0100000和0xc0100010在二进制存储的位置也相差了16个单位，所以在直接拷贝一个位置，两个地址实际的物理存储位置也相差了这么多。我们最终就读到了 `GDTR` 要加载的内容，我们在将 `cr0` 的最低一位置1，最后就开启了保护模式，ljmp更新 `cs` 寄存器，将选择子写入段寄存器。之后我们的访问，如果是一个取指，取的地址是CS:EIP，在平坦模式下，就是访问EIP我们的EIP就是从0x100000开始执行的EIP和物理内存的分布相符合，可以正常执行；主要是数据访问，数据的默认链接在了高地址，这些全局变量必须通过 va_to_pa进行转换，由于栈内的局部变量是通过和堆栈指针的相对位置进行访问的，当然不受影响。

```
void init() {
#ifdef IA32_PAGE
    /* We must set up kernel virtual memory first because our kernel thinks it
     * is located at 0xc0100000, which is set by the linking options in
Makefile.
     * Before setting up correct paging, no global variable can be used. */
    init_page();
    //if((uint32_t)init_cond > 0xc0000000)
    /* After paging is enabled, transform %esp to virtual address. */
    asm volatile("addl %0, %%esp" : : "i"(KOFFSET));
#endif
    /* Jump to init_cond() to continue initialization. */
    asm volatile("jmp *%0" : : "r"(init_cond + 0xc0000000));
    panic("should not reach here");
}

/* Initialization phase 2 */
void init_cond() {
#ifdef IA32_PAGE
    /* Initialize the memory manager. */
```

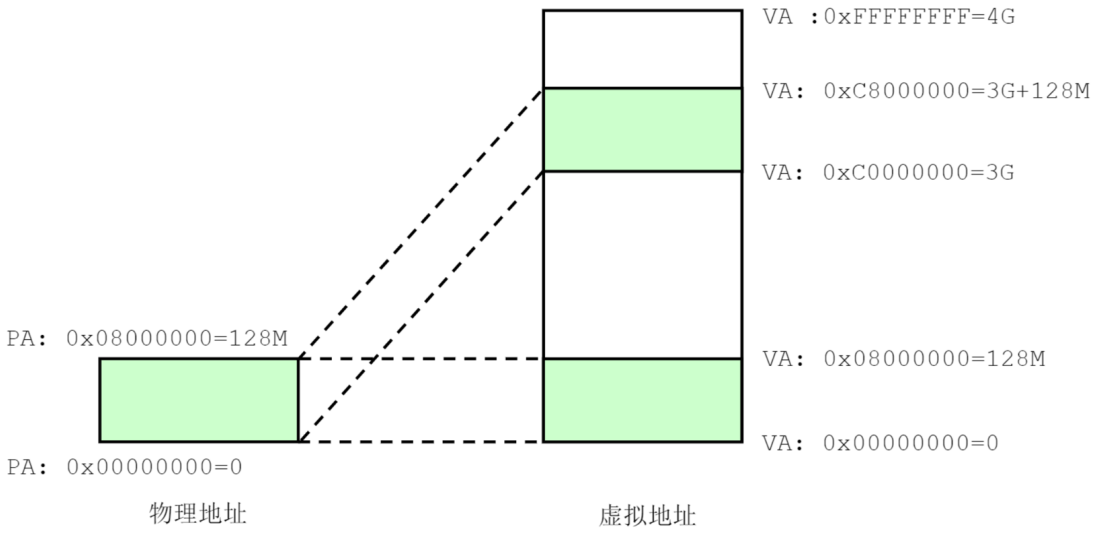
```
    init_mm();
#endif
    /* Load the program. */
    uint32_t eip = loader();

#ifdef IA32_PAGE
    /* Set the %esp for user program, which is one of the
     * convention of the "advanced" runtime environment. */
    asm volatile("movl %0, %%esp" : : "i"(KOFFSET));
#endif
    /* Keep the `bt' command happy. */
    asm volatile("movl $0, %ebp");
    asm volatile("subl $16, %esp");
    //write_cr3(get_ucr3());
    /* Here we go! */
    ((void(*)(void))eip)();

    panic("should not reach here");
}
```

现在跳转到了init，只考虑和kernel启动中PA3的页表相关部分，从大体来看：

- 我们建立了页表映射，从而安全跳到虚拟地址
- 建立用户进程的虚拟地址空间
- 加载用户进程
- 跳转到用户进程，开始我们程序的执行

```
void init_page(void) {
    CR0 cr0;
    CR3 cr3;

    PDE *pdir = (PDE *)(kpdir - 0xc0000000);
    PTE *ptable = (PTE *)(kptable - 0xc0000000);

    uint32_t pdir_idx;
    /* make all PDEs invalid */
    memset(pdir, 0, NR_PDE * sizeof(PDE));

    /* fill PDEs */
    for (pdir_idx = 0; pdir_idx < 32; pdir_idx ++) {
        pdir[pdir_idx].val = ((uint32_t)ptable)|0x7;
        pdir[pdir_idx + 768].val = ((uint32_t)ptable)|0x7;
        ptable += NR_PTE;
    }

    /* fill PTEs */
    asm volatile ("std;\
     1: stosl;\
        subl %0, %%eax;\
        jge 1b" : :
        "i"(PAGE_SIZE), "a"((PHY_MEM - PAGE_SIZE) | 0x7), "D"(ptable - 1));

    /* make CR3 to be the entry of page directory */
    cr3.val = 0;
    cr3.page_directory_base = ((uint32_t)pdir) >> 12;
```

```
    write_cr3(cr3.val);

    /* set PG bit in CR0 to enable paging */
    cr0.val = read_cr0();
    cr0.paging = 1;
    write_cr0(cr0.val);
}
```



我们总共128MB的物理内存地址空间，需要考虑要用多上的页表，几项的页目录才能满足要求，128MB = $2^{27}B$，一个页帧4KB，我们一共需要 32K 个，一个页表有1K个，所以总共需要32个页表，每个页表都塞满1024个页表项。我们将 `0~31` 和 `768~799` 映射到相同的物理地址，一部分是内核要用的高位虚拟地址，一部分内核实际的物理地址保证内核loader的正常运行。

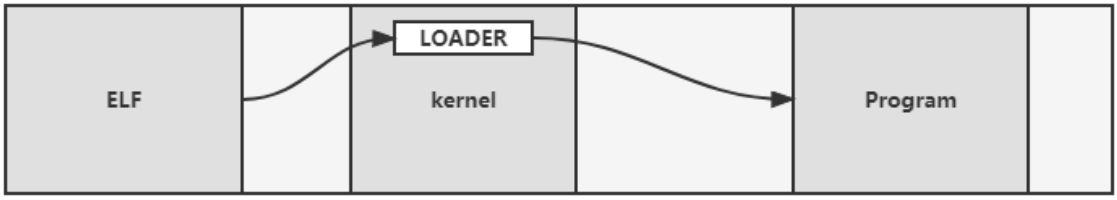
代码中对这两部分的页目录进行了映射，选取的页表为相邻的32个页表。接下来是一段内联汇编，用于直接操作汇编代码完成一些细致的操作，首先是可以看到 `std` 保证了接下的操作的方向是沿着地址低方向走的，之后不断将 `eax` 中的值转移到页表项的地址中，仍旧是高位页表项映射到高位地址，依次映射。

最后只用设置cr3寄存器设置页目录的地址（虚拟地址），开启paging启用页表，就可以通过页表正常访问了。之后设置esp的地址，进行一次跳转，跳转到高位地址空间。

```
void init_mm() {
    PDE *kpdir = get_kpdir();

    /* make all PDE invalid */
    memset(updir, 0, NR_PDE * sizeof(PDE));
    /* create the same mapping above 0xc0000000 as the kernel mapping does */
    memcpy(&updir[KOFFSET / PT_SIZE], &kpdir[KOFFSET / PT_SIZE],
           (PHY_MEM / PT_SIZE) * sizeof(PDE));
    ucr3.val = (uint32_t)va_to_pa((uint32_t)updir) & ~0xfff;
}
```

开始转向用户进程，我们拷贝内核的页目录，使我们在用户进程仍然能通过系统调用访问内核空间。

接下来就是加载用户要运行的 ELF 文件的时候了，已经开启了分页，kernel需要负责为其在用户空间分配。做的就是将用户进程的虚拟地址映射到一个物理地址，并加载到对应的物理地址上，最后做一个跳转。




```
for (i=0 ;i< e_phnum; i++)
{
        ph = (void*)buf + e_phoff + i*e_phensize;
        if(ph->p_type == PT_LOAD)
        {

            uint32_t  p_vaddr = ph->p_vaddr;
            uint32_t p_offset = ph->p_offset;
            uint32_t p_filesz = ph->p_filesz;
            uint32_t p_memsz = ph->p_memsz;

            uint8_t* addr = (uint8_t*) mm_malloc(p_vaddr,p_memsz);
            printk("addr:%x vaddr:%x p_offset:%x p_memsz:%d
p_filesz:%d\n",addr,p_vaddr,p_offset,p_memsz,p_filesz);

            ide_read(data,p_offset + ELF_OFFSET_IN_DISK,p_filesz);
            memcpy(addr,data,p_filesz);
        }

#ifdef IA32_PAGE
            /* Record the program break for future use. */
            extern uint32_t brk;
            uint32_t new_brk = ph->p_vaddr + ph->p_memsz - 1;
            if(brk < new_brk) { brk = new_brk; }
#endif

}
```

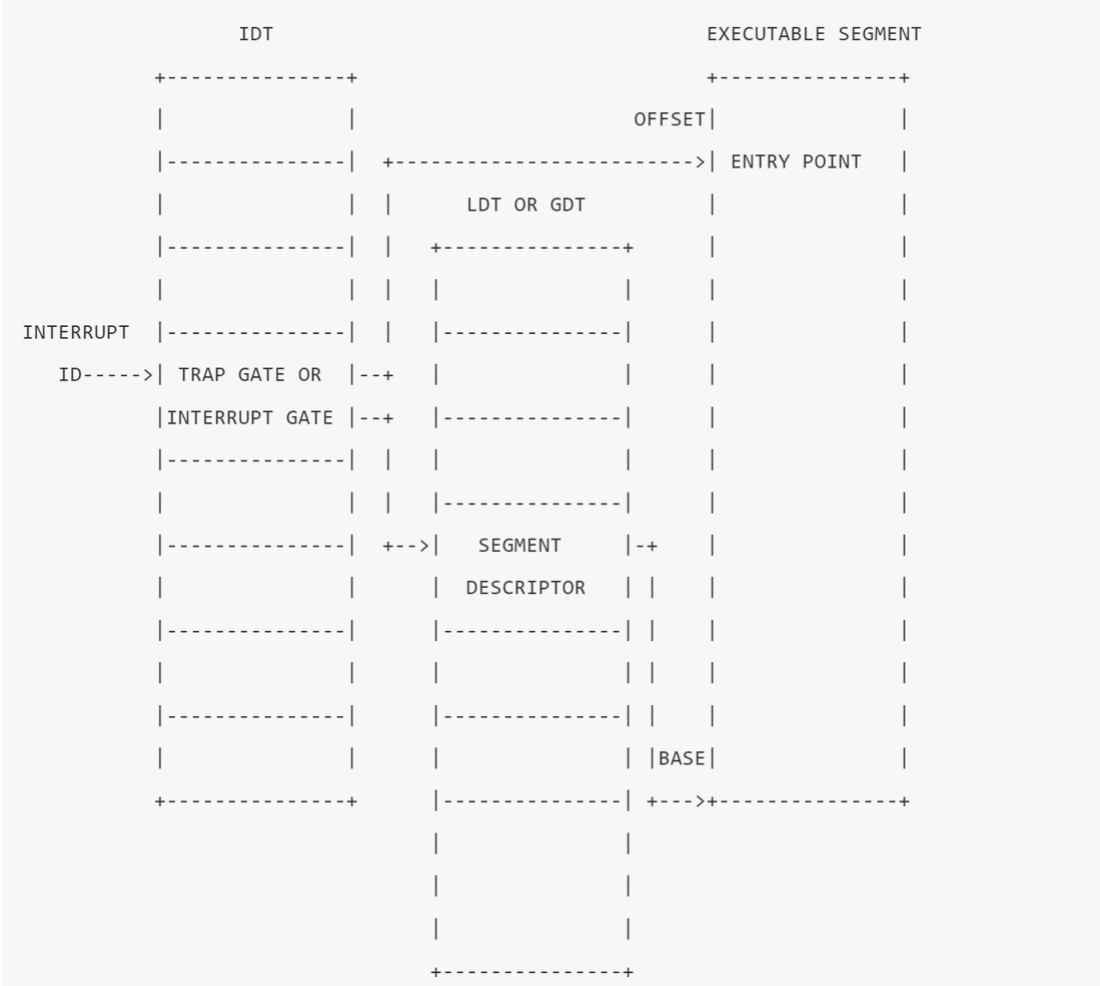
program header 是个结构体数组，那么我们可以遍历它，将属性为 `LOAD` 的做一个加载操作，具体如下，我们首先在得到 `entry` 中说明的虚拟地址，现在物理内存的分配情况，建立映射，这样在低于0xc0000000的空间建立映射到页目录中的低768项。得到映射的物理地址，将ELF文件通过IDE直接从模拟磁盘读出来，让后将其拷贝到对应的模拟内存中。

最后可以看到以new_brk，意思是我们大堆空间的申请起始地址设为在ELF文件之后，用于仙剑中的malloc实现。还有开启用户的页目录到cr3中，准备跳转到用户进程。同时我们完成kernel，就可以通过kernel启动 testcase， `make test`j 就可以自动测试。

PA4 中断和I/O

PA4开头就介绍了系统调用，用户进程通过寄存器将参数依次传入，通过系统调用触发一次异常，让后陷入内核，而这个异常是通过 `int 0x80` 实现的。紧接着是介绍触发异常之后都发生了什么，用户进程的状态为A，触发异常之后，陷入内核，内核跳转到操作系统预先设置号的异常处理代码，结束之后恢复状态A。而A的状态硬件负责保存的只有EIP,EFLAGS,CS。其他的部分将通过操作系统来进行保存。

中断和系统调用



IA32通过中断门来实现中断，中断触发后硬件会自动找到中断函数的入口，这个过程为将IDT的基址存到IDTR，一旦中断发生，中断号将作为 index 去找到对应的中断门描述符，之后通过中断门中的 selector 从GDT找对应的基址，最后通过中断门描述符中的offset 找到相对于基址的函数入口，执行中断服务例程。

接下来是一个简单的 `Sys_write` 系统调用的实现，现在 Nemu 没有实现一些IO的模拟，这里将通过Nemu本身的异常指令实现这个系统调用。这里最主要的实现就是通过 `jmpbuf` 实现 `int` 的功能，分析 `int` 如何实现才比较合理，中断发生后存储必要的上下文我们要转向中断服务例程的入口地址， 如果是只是write系统调用将没有什么区别，一条指令执行完后，马上就会到了exec函数，但如果是已经调用函数到很深的位置，比如页异常，那么跳出将是一个很麻烦的问题，异常对所涉及到可能产生异常的硬件需要增加一个异常判断。

调用栈分析在恢复环境，获取参数中很关键，下面是一个中断产生时栈标准的结构：

eflags
cs
error code
irq
eax
ecx
edx
ebx
esp
ebp
esi
edi

最后又一个 `push %esp` 这个操作用于存入当前堆栈的地址作为一个参数，该参数可以理解为一个结构体指针，指向了这个陷阱帧结构，于是在中断服务函数中，我们就可以直接使用着压入的数据，实现参数的传递。

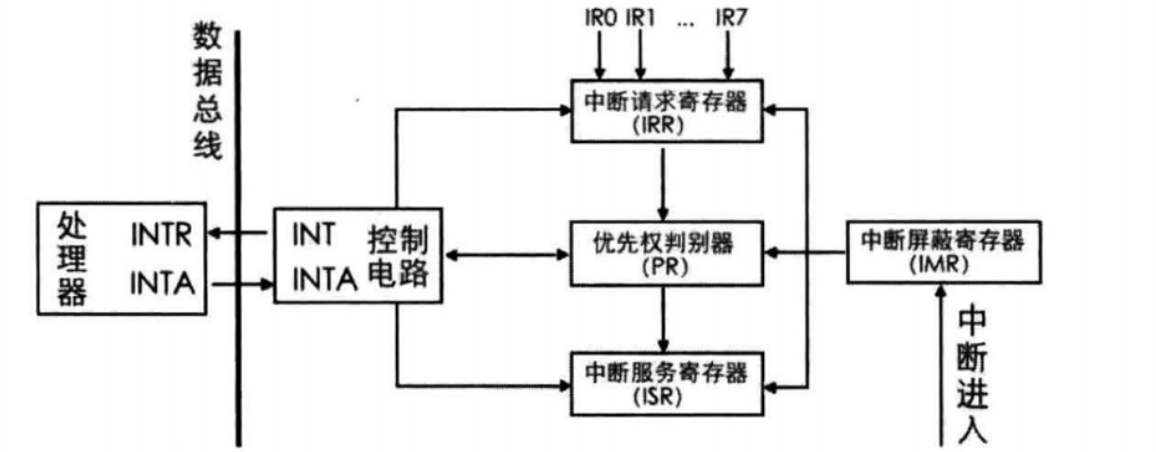
进入irq_handler后，一旦发现 `irq` 为0x80于是就知道要执行一次 system_call ,进入执行后 `eax` 最为system_call的参数，了解到这时一个 write 系统调用，于是主动触发一次 `0xd6` 这时一个异常指令，再次通过寄存器传参供异常函数识别并执行打印的操作。

```
len = cpu.edx;
i = 0;
for(;i<len;i++)
{
    char temp = swaddr_read(cpu.ecx + i,1,DS);
    buf[i] = temp;
}
printf("%s",buf);
print_asm("int 0x80");
break;
```

Nemu遇到这条指令并执行，通过`ecx`指明的地址使用`printf`进行系统调用。

设备I/O

8259



8259用于中断优先级的管理，一旦一个外部中断来临的时候，会在8259中留下一个记录，优先级判断器会通过比较优先级得出最高优先级的中断，向CPU触发一个中断请求，最后CPU取走中断号，执行中断。CPU将在每一条指令执行结束时，检测8259上是否有中断到来，一旦中断到来，就直接执行中断例程。

```
void i8259_raise_intr(int n) {
    assert(n >= 0 && n < 16);
    if(n < 8) {
        master.IRR |= MASK(n);
    }
    else {
        n -= 8;
        slave.IRR |= MASK(n);
        master.IRR |= MASK(2);

        slave.highest_irq = ffo_table[slave.IRR & ~slave.IMR];
    }

    master.highest_irq = ffo_table[master.IRR & ~master.IMR];

    do_i8259();
}
static const char ffo_table[] = {
     -1, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
     4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

每一个可以触发8259中断的外部设备在想触发一个中断时都会执行这个函数，Nemu中的8259固定采用IR2级联的方式实现，当让也不知编程配置，8259的配置在Nemu就是固定的。首先在Nemu中外部中断号一定是 `0 ~ 16` 。如果小于8说明中断不是从级联的芯片上到来的，在主片的IRR记录这个位的到来。而如果中断号大于等于8，那么可以确认是分给从片的中断号，我们中断信息到主片、从片的IRR信息上，接下来是一个ffo_table，ffo_table中使用了打表的手法。如果将一个8259芯片上的信号设为一个状态的话，从没有一个信号全0到8个信号全1，一种有256个值，除了全0代表没有中断到来，其他任何一种情况哪一个信号优先级最高都是确定的，我们将这种对应打成一个表就又有了ffo_table。于是我们就得到了主片和从片上分别的最高特权级：

- 当主片上最高特权级小于2时，特权最高的在主片上
- 当主片上最高特权大于等于2时，特权级最高的在从片上
- 选出最高特权级，输出中断号

为了讲解接下来的部分，必须先分析Nemu对于端口I/O的模拟，用一个数据结构记录端口映射关系，在每次进行IO读写时，才会待用设备提供的回调函数，对于显存还提供了内存映射IO的功能。具体如下：

0	1	2	3	4	5	6	...	...	65535

代码中申请了这样一个Byte数组，用于模拟CPU的端口。同时支持I/O端口的注册

```
typedef struct {
    ioaddr_t low;
    ioaddr_t high;
    pio_callback_t callback;
} PIO_t;
```

PIO_t 用于端口注册的结构，使用如下函数进行一次注册

```
void* add_pio_map(ioaddr_t addr, size_t len, pio_callback_t callback) {
    assert(nr_map < NR_MAP);
    assert(addr + len <= PORT_IO_SPACE_MAX);
    maps[nr_map].low = addr;
    maps[nr_map].high = addr + len - 1;
    maps[nr_map].callback = callback;
    nr_map ++;
    return pio_space + addr;
}
```

Nemu中准备了8个PIO_t结构用于注册IO端口，每次注册找一个空闲的PIO_t , low 和 high 指明了IO端口的左右范围，最后是每当读写的该端口时该执行什么回调函数，用于模拟真实读写IO端口时IO端口的响应，最后返回了IO端口的起始地址，用数据的传递。

内存映射IO也是同理，只不过内存映射IO的空间往往比较大，所以直接将指针赋值给了 MMIO_t 内部的结构，同时访问这部分IO时也会自动执行回调函数。以mmio_read为例，每次访问mmio， 都会在执行mmio_read 或者 mmio_write 二者之一，当让如果判断出这个部分的内存被映射到IO，将不会访问 Cache 而直接访存，这一部分内容要在Nemu的访存部分补充。可以看见一旦执行了mmio_read,直接读写了mmio对应的存储空间，并执行了回调函数，false 代表了这时一个读操作。

```c
typedef struct {
    hwaddr_t low;
    hwaddr_t high;
    uint8_t *mmio_space;
    mmio_callback_t callback;
} MMIO_t;

uint32_t mmio_read(hwaddr_t addr, size_t len, int map_NO) {
    assert(len == 1 || len == 2 || len == 4);
    MMIO_t *map = &maps[map_NO];
    uint32_t data = *(uint32_t *)(map->mmio_space + (addr - map->low))
        & (~0u >> ((4 - len) << 3));
    map->callback(addr, len, false);
    return data;
}
```

IDE

模拟磁盘的功能，把传入的ELF文件当作一个磁盘，之后以扇区为单位进行读写，kernel通过驱动可以进行文件的读写。这一个模块需要分为两个部分进行理解，一个时Nemu的磁盘实现，另一个时kernel中的IDE驱动实现。首先开Nemu中的磁盘实现，这里只考虑直接操作磁盘（即直接读写输入的ELF文件），而不去考虑DMA。

```c
void ide_io_handler(ioaddr_t addr, size_t len, bool is_write) {
    assert(byte_cnt <= 512);
    int ret;
    if(is_write) {
        if(addr - IDE_PORT == 0 && len == 4) {
            /* write 4 bytes data to disk */
            assert(ide_write);
            ret = fwrite(ide_port_base, 4, 1, disk_fp);
            assert(ret == 1);

            byte_cnt += 4;
            if(byte_cnt == 512) {
                /* finish */
                ide_port_base[7] = 0x40;
            }
        }
        else if(addr - IDE_PORT == 7) {
            if(ide_port_base[7] == 0x20 || ide_port_base[7] == 0x30) {
                /* command: read/write */
                sector = (ide_port_base[6] & 0x1f) << 24 | ide_port_base[5] <<
16
                    | ide_port_base[4] << 8 | ide_port_base[3];
                disk_idx = sector << 9;
                fseek(disk_fp, disk_idx, SEEK_SET);

                byte_cnt = 0;

                if(ide_port_base[7] == 0x20) {
                    /* command: read from disk */
                    ide_write = false;
                    ide_port_base[7] = 0x40;
```

```
                    i8259_raise_intr(IDE_IRQ);
                }
                else {
                    /* command: write to disk */
                    ide_write = true;
                }
            }
        }
    }
    else {
        if(addr - IDE_PORT == 0 && len == 4) {
            /* read 4 bytes data from disk */
            assert(!ide_write);
            ret = fread(ide_port_base, 4, 1, disk_fp);
            assert(ret == 1 || feof(disk_fp));

            byte_cnt += 4;
            if(byte_cnt == 512) {
                /* finish */
                ide_port_base[7] = 0x40;
            }
        }
    }
}
```

```
考虑一个一个执行流程如下：
    write:
        1.确定要写的区块地址，out(IDE_PORT + 3, low_addr)
        out(IDE_PORT+4, mid_addr), out(IDR_PORT+5,high_addr),
out(IDE_PORT+6,left)
        2.issue_write, out(IDE_PORT + 7, 0x30);
        3.写数据,out(IDE_PORT,data)
    read:
        1.确定要读的区块地址，out(IDE_PORT + 3, low_addr)
        out(IDE_PORT+4, mid_addr), out(IDR_PORT+5,high_addr),
out(IDE_PORT+6,left)
        2. issue_read, out(IDE_PORT + 7, 0x20),设置7号口为0x40，同时发起一个中断
        3. read,in(IDE_PORT)
```

kernel 驱动中最底层是一个disk_do_read 和一个 disk_do_write 两个函数实现了我们如上所说的执行流程，通过写区块地址、issue、读写磁盘三个步骤对磁盘的调用规则进行了封装，实现了读区块和写区块的功能。再往上一层就是buffer层的抽象，这里对我们输入的磁盘块进行了缓存，缓存通过直接映射决定区块映射到哪一块的缓存中，当区块就在缓存中就不需要访问磁盘，但是如果映射后发现对应的缓存中不是该区块也不为空，那么就要根据占用该缓存的区块是否是“脏的”，决定是否写回，之后将要的区块写入缓存。这一层向上提供了读一个字节和写一个字节的抽象。

最后我们使用的函数就是ide_read 和 ide_write 通过文件的偏移直接读写磁盘，同时注册了一个中断函数，使得0（实际的irq为1000）中断发生时直接将所有区块从buffer中写回。

VGA

这个模块VGA的功能，在IO端口中注册用于修改调色板的两个端口，同时从 0xa0000 开始的一段内存作为video memory，通过修改 video memory 的内容，修改虚拟屏幕的内容。VGA可以说是Nemu外设中较为复杂的一个部分，这个部分涉及了很多SDL的知识，也涉及了很多VGA的知识，其实就是用如何用SDL模拟一个VGA的显示器。

- 先看下面的两个函数第二个函数是一个ctrl口，它注册了两个端口 `0x3d4` 和 `0x3d5` ，如果向 `0x3d4` 中写入一个控制寄存器号，那么在 `0x3d5` 我们就可以读出那个控制寄存器的值；如果向 `0x3d5` 中写入一个值，那么这个值将传给 `0x3d4` 中指明的控制寄存器。
- 第一个函数用于更新调色板，只有写的时候才会调用回调函数
  - 第一种是写 `VGA_DAC_WRITE_INDEX` 用于设置将要修改的调色板数组的下标。
  - 第二种是写 `VGA_DAC_DATA` 用于实际更新调色板，`(((void *)color_ptr - (void *)&screen->format->palette->colors) & 0x3) == 3` 意味着调色板4bytes 的更新已经到第四个，最后如果这已经将256个调色板单元全部更新完毕那么，我们将这个调色板注册到我们申请的表面上，这两个表面一个作为逻辑屏幕，一个作为实际屏幕。

```
/* palette contains 256 color used to show on screen, vga_dac_port figure out the
index of it*/
/*
这个函数用于修改调色板，操作时先设置要修改的调色板的坐标，如果修改的是四个颜色域中的最后一个，判断是否已经是调色板的最后一个index
最后设置调色板到逻辑屏幕和物理屏幕上。
*/
void vga_dac_io_handler(ioaddr_t addr, size_t len, bool is_write) {
    static uint8_t *color_ptr;
    if(addr == VGA_DAC_WRITE_INDEX && is_write) {
        color_ptr = (void *)&palette[ vga_dac_port_base[0] ];
    }
    else if(addr == VGA_DAC_DATA && is_write) {
        *color_ptr++ = vga_dac_port_base[1] << 2;
        if( (((void *)color_ptr - (void *)&screen->format->palette->colors) &
0x3) == 3) {
            color_ptr ++;
            if((void *)color_ptr == (void *)&palette[256]) {
                SDL_SetPalette(real_screen, SDL_LOGPAL | SDL_PHYSPAL, (void
*)&palette, 0, 256);
                SDL_SetPalette(screen, SDL_LOGPAL, (void *)&palette, 0, 256);
            }
        }
    }
}

/*如何操作控制寄存器?
1.向VGA_CRTC_INDEX中写入control register的编号，VGA_CTRL_DATA中就可以获取该control寄存器的值;
2.向VGA_CTRL_DATA中写入数据，根据VGA_CRTL_INDEX的值，将对应的control register设为输入的数据
*/
void vga_crtc_io_handler(ioaddr_t addr, size_t len, bool is_write) {
    if(addr == VGA_CRTC_INDEX && is_write) {
        vga_crtc_port_base[1] = vga_crtc_regs[ vga_crtc_port_base[0] ];
    }
    else if(addr == VGA_CRTC_DATA && is_write) {
        vga_crtc_regs[ vga_crtc_port_base[0] ] = vga_crtc_port_base[1] ;
    }
}
```

屏幕更新原理

已经介绍了屏幕跟新是根据显存对应部分是否 “脏了” 判断是否进行跟新，我们每写显存的时候会调用回调函数将对应的显存行设置为 "脏"，之后每次update_device 最会调用 显示器更新，根据一行的数据用显存的内容draw_pixel到逻辑屏幕上，一旦更新成功将逻辑屏幕更行到真实屏幕上，至于为什么跟新4个点是因为屏幕的像素个数是显存个数的4倍。

```
void do_update_screen_graphic_mode() {
    int i, j;
    uint8_t (*vmem) [CTR_COL] = vmem_base;
    SDL_Rect rect;
    rect.x = 0;
    rect.w = CTR_COL * 2;
    rect.h = 2;

    for(i = 0; i < CTR_ROW; i ++) {
        if(line_dirty[i]) {
            for(j = 0; j < CTR_COL; j ++) {
                uint8_t color_idx = vmem[i][j];
                draw_pixel(2 * j, 2 * i, color_idx);
                draw_pixel(2 * j, 2 * i + 1, color_idx);
                draw_pixel(2 * j + 1, 2 * i, color_idx);
                draw_pixel(2 * j + 1, 2 * i + 1, color_idx);
            }
            rect.y = i * 2;

            //map logical screen on physic screen
            SDL_BlitSurface(screen, &rect, real_screen, &rect);
        }
    }
    SDL_Flip(real_screen);
}
```

时钟

代码中的定时器是虚拟定时器，它只工作在用户态，如果输出大量的IO，计时的速度将变得十分缓慢。

信号的发送

发送信号的主要函数有：kill、raise、sigqueue、alarm、setitimer、abort。

1. kill，原型为int kill(pid_t pid, int signo), signo是信号值。
2. raise, 原型为int raise(int signo), 向进程本身发送信号。
3. sigqueue，原型为int sigqueue(pid_t pid, int sig, const union sigval val),前两个参数为pid和发送的信号值，第三个指定了信号传递的参数。在调用sigqueue会将传递的参数拷贝到信号处理函数。
4. alarm，原型为unsigned int alarm(uisigned int seconds), 专门为SIGALRM信号设定，执行的seconds后向进程本身发送SIGALRM信号。进程调用alarm，任何之前调用的alarm都将无效。而返回值，如果在调用alarm前，进程已经调用了闹钟，则返回上一个闹钟的剩余时间，否则返回0。
5. setitimer, 原型为int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue),第二个参数是itimerval的一个实例，第三个参数可以忽略。第一参数可以指定时钟的类型：

```
1. ITIMER_REAL: 设定绝对时间，经过指定的时间后，内核发送SIGALRM信号给本进程。
2. ITIMER_VIRTUAL: 设定程序执行时间，经过指定的时间后，发送SIGALRM信号给本进程。
3. ITIMER_PROF: 设定进程执行以及内核因本进程而消耗的时间和，经指定的时间后，内核将发送
ITIMER_VIRTUAL信号给本进程
```

1. abort, 原型为void abort(void), 向进程发送SIGABORT信号，默认情况下进程会退出。

信号的安装

如果进程要处理一个信号，那么就要在进程中安装这个信号，安装信号主要确定信号值和该信号对信号的动作之间的映射。信号的安装主要有两个函数signal()、sigaction():

1. signal, 原型为 void signal(int signnum, void (*handler)(int))(int), 为signnum安装handler操作。
2. sigaction，原型为int sigaction(int signnum, const struct sigaction *act, struct sigaction *oldact)。用于定义进程就收特定信号后的行为，第二信号最为重要，其中包含了对特定信号的处理，信号传递的信息，信号处理过程中应屏蔽的函数等等。

```
struct sigaction
{
    union
    {
        __sighandler_t _sa_handler;
        void (* _sa_sigaction)( int, struct siginfo *, void *);
    }_u;

    sigset_t sa_mask;
    unsigned long sa_flags;
}

1. 在联合结构体_u中，由_sa_handler以及*_sa_sigaction指定信号关联函数，即用户指定的信号处理函数。由_sa_handler指定的处理函数只能由一个参数，即信
号值。而由_sa_sigaction指定的处理函数可以带三个参数，它指定一个3个参数的信号处理函数，第一个参数为信号值，第三个参数没有用，第二个参数的是指向
siginfo_t结构的指针，结构体中包含信号携带的数据值。

siginfo_t {
    int si_signo;       //信号值
    int si_errno;       //error值
    int si_code;        //信号产生的原因
    pid_t si_pid;       //发送信号的进程ID,对于kill(2),实时信号以及SIGCHLD有意义
    uid_t si_uid;       //发送信号进程的真实用户ID，对于kill(2)，实时信号以及SIGCHLD有意
义
    int si_status;      //退出状态，对SIGCHLD有意义
    clock_t si_utime;   //用户消耗的时间，对SIGCHLD有意义
    clock_t si_stime;   //内核消耗的时间，对SIGCHLD有意义
    sigval_t si_value;  //信号值，对于所有实时信号有意义，是一个联合结构体

    void* si_addr;      //触发fault的内存地址，对于SIGILL,SIGFPE,SIGSEGV,SIGBUS 信号
有意义
    int si_band;        //对于SIGPOLL有意义
    int si_fd;      //对于SIGPOLL有意义
}

si_value是一个联合结构体，结构如下：
union sigval{
    int sival_int;
    void *sival_ptr;
```

}

前面讨论的sigqueue发送信号时，sigqueue的第三个参数就是sigval联合结构体，当调用sigqueue时，该数据结构中的数据接拷贝到信号处理函数的的第二
个参数中。

2. sa_mask指定在信号处理过程中，哪些信号应当被阻塞。缺省情况下当前信号本身因该被阻塞，防止信号的嵌套发送，除非指定了SA_NIDEFER或者SA_NOMASK
标志位。

3.sa_flags中包含了许多的标志位，包括SA_NODEFER以及SA_NOMASK标志位。另一个比较重要的标志位是SA_SIGINFO，当设定了该标志位时，表示该信号附带
的参数可以被传递到信号处理函数中，因此，在sigaction结构体中的sa_sigaction指定处理函数，而不因该为sa_handler指定信号处理函数，否则，这个设置
将变得含无意义。

```
struct sigaction s;
memset(&s, 0, sizeof(s));
s.sa_handler = timer_sig_handler;
ret = sigaction(SIGVTALRM, &s, NULL);
Assert(ret == 0, "Can not set signal handler");

it.it_value.tv_sec = 0;
it.it_value.tv_usec = 1000000 / TIMER_HZ;
ret = setitimer(ITIMER_VIRTUAL, &it, NULL);
```

```
static void timer_sig_handler(int signum) {
    jiffy ++;
    timer_intr();

    device_update_flag = true;
    if(jiffy % (TIMER_HZ / VGA_HZ) == 0) {
        update_screen_flag = true;
    }

    int ret = setitimer(ITIMER_VIRTUAL, &it, NULL);
    Assert(ret == 0, "Can not set timer");
}
```

这里设置了一个虚拟时钟，只统计在用户态的运行时间，同时时钟周期为100HZ，每10ms 出现一个闹钟信号，我们已经为闹钟信号注册了一个函数，每次闹钟信号出现，同会执行timer_sig_handler。每次时钟来临的时候，直接触发一次8259中断，然后根据显示否"脏了"，更新显示器（40ms一次），同时设置下一个闹钟信号。

串口

每当CPU往串口的数据寄存器输入数据时，串口将输入会发送到主机的标准输出。

```
//实际上只进行一个工作将输入端口数组的数据放到stdout
void serial_io_handler(ioaddr_t addr, size_t len, bool is_write) {
    if(is_write) {
        assert(len == 1);
        if(addr == SERIAL_PORT + CH_OFFSET) {
            char c = serial_port_base[CH_OFFSET];
            /* We bind the serial port with the host stdout in NEMU. */
            putc(c, stdout);
```

```
            if(c == '\n') {
                fflush(stdout);
            }
        }
    }
}
```

键盘

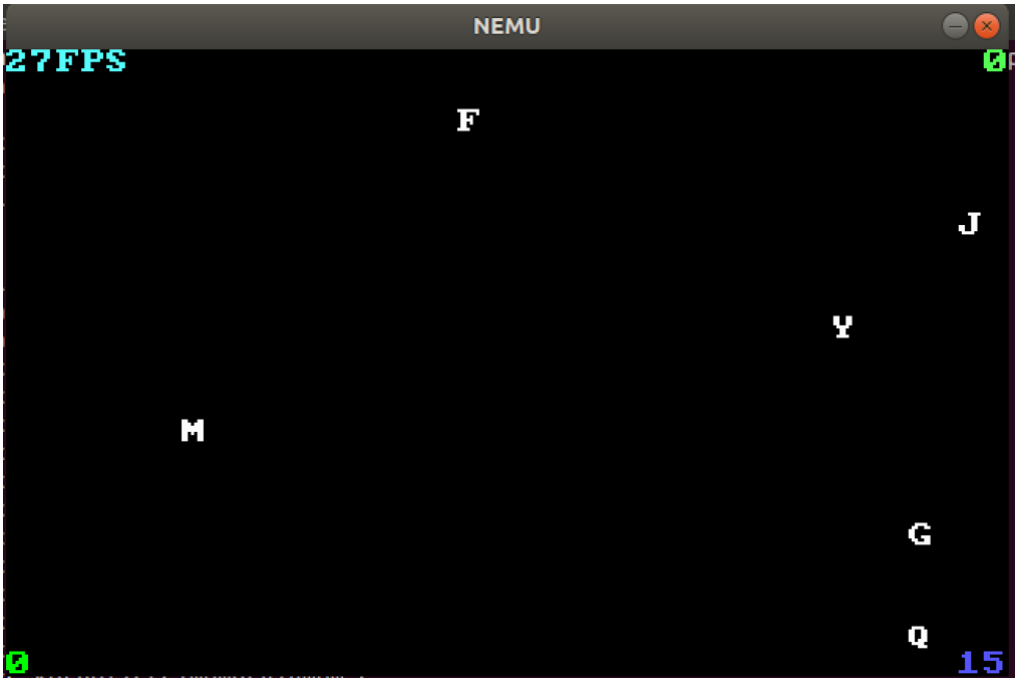
Nemu中引入SDL库，Nemu通过SDL中的函数来实现对键盘输入的检测。Nemu中并能像真实的硬件一样，键盘一敲下，键盘主动告诉CPU我被敲了。Nemu中主要两个地方执行了下面这个函数，下面这个函数用于主动地更新设备状态，通过显示器是否有“脏了”的内容跟新显示器，接下来主动扫描SDL的事件池中是否有键盘事件，从下面的while循环可以看出键盘信息的提取只有最后一个能生效，但是好在这个时间间隔是很短的。

- 这个函数在每一次执行完一条指令都会主动检查一次，思考如果没有时钟这个函数能正常执行吗？可以只需不管flag，主动检测显存内容是否"脏了"，从事件池中提取信息不依赖时钟信号，但是时钟对于游戏的实现还是有很重要的作用的。
- 第二个使用这个函数的地方在`hlt`函数，`hlt`是使CPU空转的指令，在中断来临之前CPU将什么也不做，实现这个指令很朴素地想应该是个循环`while( INTR == false)`可以等待通过宿主系统的时钟信号来主动跟新这个INTR，但是会出现一个问题就是在这个时间之内，也就是10ms之内，Nemu什么也做不了，尽管只有10ms但是为了保证设备的流畅性，在循环中也要实现一个设备检查如果有键盘中断可以直接跳出这个循环，这个对游戏很重要，事实上打字小游戏中就是通过来等待一个键盘输入，也就是说如果键盘事件没有到来的话，打字游戏将至少10ms进行一次`GAME_Loop`。

```
void device_update() {
    if(!device_update_flag) {
        return;
    }
    device_update_flag = false;

    if(update_screen_flag) {
        update_screen();
        update_screen_flag = false;
    }

    SDL_Event event;
    while(SDL_PollEvent(&event)) {
        // If a key was pressed

        uint32_t sym = event.key.keysym.sym;
        if( event.type == SDL_KEYDOWN ) {   //键盘按下
            keyboard_intr(sym2scancode[sym >> 8][sym & 0xff]);
        }
        else if( event.type == SDL_KEYUP ) { //键盘抬起
            keyboard_intr(sym2scancode[sym >> 8][sym & 0xff] | 0x80);
        }

        // If the user has Xed out the window
        if( event.type == SDL_QUIT ) {
            //Quit the program
            exit(0);
        }
```

```
    }
}
```

打字小游戏



打字游戏用简短的程序说明了一个游戏运行的基本流程：

```
void main_loop(void)
{
    ...
    while(true)
    {
        等待键盘事件或者时钟中断的到来

        关中断
        获取当前时间，系统时间
        判断游戏中的时间  是否小于  系统时间
        开中断

        跟新字母链表中的状态，被键盘击中的字母要处理

        while(但前时间 < 系统时间)
        {
            跟新字母位置
            生成新的字母
            计算帧率
        }

        更新屏幕
    }
}
```

游戏的基本框架在这关键是如何通过系统来帮助游戏的实现，答案是通过系统调用，Nemu通过0号系统中断可以注册中断服务函数，打字小游戏中注册了两个中断服务函数，来告诉中断来了时Nemu要帮游戏做什么

- 时钟中断来临时用于更新游戏系统时间。

- 键盘中断到来，更新按键按下的状态信息。

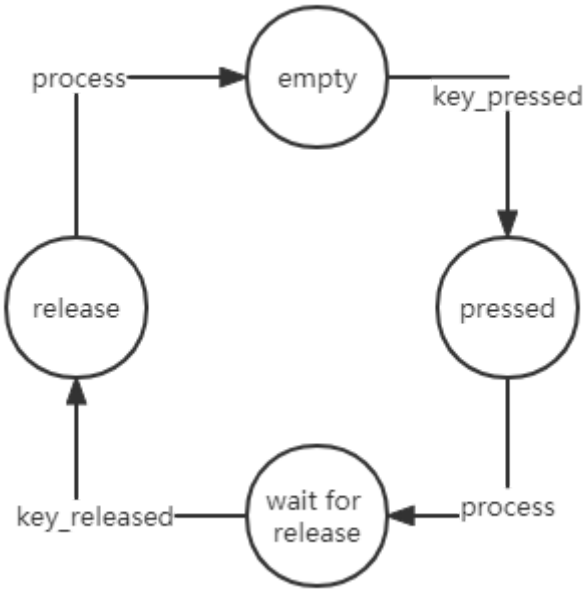
仙剑奇侠传（未成功运行）

文件系统

仙剑奇侠传的运行要求实现一个简单的文件系统，这个文件系统中的每个文件的位置被写死，同时文件的大小保持不变，文件不会增加也不会减少，大大降低了文件系统的实现难度。file_array中记录所有文件的名称，大小，在磁盘中的偏移。我们只需要维护一个文件的打开结构就可以了，这个结构中记录了文件是否被打开和当前文件指针的位置。

- open ，输入一个要打开的文件名，遍历一边file_array如果出现了和输入文件名一样的名字，说明了要找的文件，设置 index + 3 为 返回的 文件描述符 fd， 因为低三位要让个 STDIN, STDERR, STOUT。
- read ，如果fd代表的文件已经被打开的话，只要文件指针的位置加上要访问的长度不超过文件的大小，那么就通过ide_read读磁盘，并改变offset的值。
- write， 如果fd代表的文件已经被打开的话，只要文件指针的位置加上要访问的长度不超过文件的大小，那么就通过ide_write写磁盘，并改变offset。
- lseek， 有SET,CUR,END三个选项，SET可以直接设置当前的文件指针，CUR可以设置文件指针相对于当前的位置，END可以设置文件指针相对于文件尾部的位置，注意这里的指针值必须大于0 小于 文件的实际大小。
- close ，关闭一个文件。

键盘重复检测



在仙剑奇侠传中我们要防止，一个案件长按的得到不断地响应，在代码中设设置一个数组，这个数组记录了所有地使用的按键的当前的状态，`empty`、`pressed`、`wait for release`、`release`。每次按下一个按键，都会将所有的状态的empty的按键更新为`pressed`；松开一个键，如果这个键是`wait for release`变成release 。每一次process key会扫描一次状态数组将，处理变成pressed和release的状态，并调用回调函数，这个过程保证了在process key前所有的press，release最终只会被响应一次。

SDL API

Nemu上运行的仙剑奇侠传运行依赖SDL，我们需要实现SDL中的几个API使得仙剑奇侠传能够成功运行。

举一个例子，SDL中的Blit是将一个表面的一部分上的像素拷贝到另一个表面的一部分像素上，`SDL_BlitSurface` 指令了 将 src 上的 srcrect 部分映射到dst的dstrect坐标之上。

```
void SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst,
SDL_Rect *dstrect)
{
    int x, y, tmp_x, tmp_y, w, h, start_x, start_y;
    if( dstrect != NULL )
    {
        x = dstrect->x;
        y = dstrect->y;
    }
    else
    {
        x = 0;
        y = 0;
    }

    if( srcrect != NULL )
    {
        start_x = srcrect->x;
        start_y = srcrect->y;
        w = srcrect->w;
        h = srcrect->h;
    }
    else
    {
        start_x = 0;
        start_y = 0;
        w = src->w;
        h = src->h;
    }

    int i,j;
    for( i = 0; i < w; i++ )
    {
        for( j = 0; j < h; j++ )
        {
            tmp_x = x + i;
            tmp_y = y + j;
            dst->pixels[ tmp_y * dst->w + tmp_x ] = src->pixels[ (start_y + j) *
src->w + start_x + i ];
        }
    }

}
```