

南京航空航天大学《计算机组成原理II课程设计》报告

- 姓名：邵震哲
- 班级：1620204
- 学号：162020130
- 报告阶段：PA3.1
- 完成日期：2022.6.1
- 本次实验，我完成了所有内容。

目录

南京航空航天大学《计算机组成原理II课程设计》报告

目录

思考题

1. 什么是操作系统？ (5)
2. 我们不一样，吗？ (5)
3. 操作系统的实质 (5)
4. 程序真的结束了吗？ (10)
5. 触发系统调用 (10)
6. 有什么不同？ (5)
7. 段错误 (10)
8. 对比异常与函数调用 (5)
9. 诡异的代码 (5)
10. 注意区分事件号和系统调用号 (5)
11. 打印不出来？ (5)
12. 理解文件管理函数 (15)
13. 不再神秘的秘技 (5)
14. 必答题 (5)
15. git log和git branch截图 (5)

实验内容

- 实现 loader (10分)
- 添加寄存器和 LIDT 指令 (10 分)
- 实现 INT 指令 (10 分)
- 实现其他相关指令和结构体 (15 分)
- 完善事件分发和 do_syscall (15 分)
- 实现堆区管理 (10 分)
- 实现系统调用 (10 分)
- 成功运行各测试用例 (20 分)

遇到的问题及解决办法

实验心得

其他备注

思考题

1. 什么是操作系统？（5）

操作系统是控制和管理整个计算机系统的硬件和软件资源的计算机程序，合理的组织和调度计算机的工作和资源的分配，以提供给用户和其它软件方便的接口和环境，它是计算机系统中最基本的系统软件。

2. 我们不一样，吗？（5）

`nanos-lite` 在调用 AM 的API的基础上，实现了更多的功能和接口，例如文件系统、终端异常处理、加载器等。

可以看作是同等地位。

3. 操作系统的实质（5）

程序

无

4. 程序真的结束了吗？（10）

main函数执行之前，主要就是初始化系统相关资源：

- 1.设置栈指针
- 2.初始化static静态和global全局变量，即data段的内容
- 3.将未初始化部分的赋初值：数值型short, int, long等为0, bool为FALSE, 指针为NULL, 等等, 即.bss段的内容
- 4.运行全局构造器
- 5.将main函数的参数, argc, argv等传递给main函数, 然后才真正运行main函数

main函数执行之后：

- 1.获取main的返回值给操作系统
- 2.调用exit退出程序

5. 触发系统调用（10）

编写hello.c

```
const char str[] = "Hello world!\n";

int main() {
    asm volatile ("movl $4, %eax;"      // system call ID, 4 = SYS_write
                  "movl $1, %ebx;"      // file descriptor, 1 = stdout
                  "movl $str, %ecx;"    // buffer address
                  "movl $13, %edx;"     // length
                  "int $0x80");

    return 0;
}
```

运行结果

```
shaozhenzhe@Debian:~/helloproject$ make run
gcc hello.c -o ./hello
./hello
Hello world!
shaozhenzhe@Debian:~/helloproject$
```

6. 有什么不同？ (5)

与函数调用过程十分类似。函数调用时，获取调用函数的起始地址生成新的栈帧，并跳转过去，调用后回复。

可以。系统调用会根据系统调用号在 IDT 中索引，取得该调用号对应的系统调用服务程序的地址，并跳转过去。在触发系统调用前，会保护用户相关状态寄存器（EFLAGS, EIP等）到栈中，系统调用完毕后再恢复。这个过程与函数调用的过程基本一致，因此可以认为系统调用的服务程序理解为一个比较特殊的函数。

不同之处在于服务程序的工作都是硬件自动完成的，不需要程序员编写指令来完成相应的内容，遇到异常才会触发。

7. 段错误 (10)

编译的时候只会检查语法，不检查具体的指令操作，只有在执行时访问的时候访问到不访问的地方时才会报错。

通常由以下程序行为引起的：

- 访问系统数据区
- 内存越界，访问到不属于程序的内存区域（数组越界，变量类型不一致）
- 多线程程序使用了不安全的函数
- 多线程读写数据未加锁保护
- 堆栈溢出

8. 对比异常与函数调用 (5)

异常

保存寄存器、错误码 #`irq`、EFLAGS、CS、EIP，形成了 trap frame（陷阱帧）的数据结构。

函数调用

只要保存寄存器

在异常处理的时候已经切换了栈帧，所以要保存更多的信息。

9. 诡异的代码 (5)

以指向 trapframe 内容的指针（esp）作为参数，调用 trap 函数。把 eip 作为入口参数传进去，然后在执行 `irq_handle` 这个函数之前，通过 `pusha`，在栈帧中形成了 `_RegSer` 这个结构体，把 eip 作为一个结构体的起始地址，通过成员 `irq` 来分发事件。

10. 注意区分事件号和系统调用号 (5)

事件号：标明一个未实现的系统调用事件的编号。

系统调用号：在识别出系统调用事件后，从寄存器中取出系统调用号和输入参数，根据系统调用号查找系统调用分派表，执行相应的处理函数，并记录返回值。

11. 打印不出来？（5）

`printf` 打印是行缓冲，读取到的字符串会先放到缓冲区里，直到一行结束或者整个程序结束，才输出到屏幕，因为我们打印的字符串一行没有结束，因此没有输出，先执行到后面的报错部分了。

因此，只需要在字符串后面加上 `\n` 就表明一行结束，在执行到错误前把内容输出即可。

```
#include <stdio.h>
int main(){
    int *p = NULL;
    printf("I am here!\n"); //在这里换行
    *p = 10; // giving value to a NULL address will cause segmentation fault
    return 0;
}
```

12. 理解文件管理函数（15）

- `fs_open` :

按照文件名在 `file_table` 里面搜索，如果找到了，则把该文件的读写指针标为0并返回下标；如果找不到就报错并返回-1。

- `fs_read` :

根据 `fd` 找文件开始的位置和剩余的大小，调用函数读取指定位置和长度的内容，最后更新读写指针。

- `fs_write` :

根据 `fd` 找文件开始的位置和剩余的大小，调用函数写入指定位置和长度的内容，最后更新读写指针。

- `fs_lseek` :

根据 `fd` 找文件开始的位置和剩余的大小，然后根据 `whence` 来更新 `new_offset`。若更新后，`new_offset` 小于0或者大于文件大小，则把其置为0或文件大小并返回。

- `fs_close` :

返回0

13. 不再神秘的秘技（5）

应该是变量类型没注意，造成在某些特定情况下，会出现溢出现象。

14. 必答题（5）

- 存档读取

`PAL_LoadGame()` 先打开指定文件然后调用 `fread()` 从文件里读取存档相关信息（其中包括调用 `nanos.c` 里的 `_read()` 以及 `syscall.c` 中的 `sys_raed()`），随后关闭文件并把读取到的信息赋值（用 `fs_write()` 修改），接着使用AM提供的 `memcpy()` 拷贝数据，最后使用 `nemu` 的内存映射I/O修改内存。

- 更新屏幕

`redraw()` 调用 `ndl.c` 里面的 `NDL_DrawRect()` 来绘制矩形，`NDL_Render()` 把VGA显存抽象成文件，它们都调用了 `nanos-lite` 中的API，最后 `nemu` 把文件通过I/O接口显示到屏幕上。

15. git log和git branch截图 (5)

git log 截图

```
shaozhenzhe@Debian: ~/ics2022
fb0b43f (HEAD -> pa3) PA3.1 finished
91da6fd > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:50:57 up 4:36, 2 users, load average: 0.13, 0.42, 0.49 a3474dfb10c50a34a051329d2ab1468c221b634b
962bfdc > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:46:53 up 4:32, 2 users, load average: 0.58, 0.66, 0.56 a26c1240dd1d1eb9f71fd89eb912089e8c06c3d
8144554 > compile 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:46:53 up 4:32, 2 users, load average: 0.58, 0.66, 0.56 elb39e1880d6f53da2b8ca348677ef7479778e06
94972e2 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:43:13 up 4:28, 2 users, load average: 0.18, 0.24, 0.42 76e9b42a3d67e36a2fb28182e8balf9ee6339ee
49df9c8 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:42:31 up 4:28, 2 users, load average: 0.26, 0.26, 0.43 7687e9579f3428d5e9a6bf1225cfa175826829
8dee291 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:41:04 up 4:26, 2 users, load average: 0.08, 0.21, 0.43 842c9b61a48deec3d67a8eef41bc3dd916e08cbf
26957ef > compile 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:41:04 up 4:26, 2 users, load average: 0.08, 0.21, 0.43 59da755c49d368f65569f27821e62bc39e0f43a
fd9591c > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:33:09 up 4:18, 2 users, load average: 0.57, 0.85, 0.69 5cd6bf7f9a08e9c0a37c3e7b212dc0b73b2133a1
d43df4b > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:32:00 up 4:17, 2 users, load average: 0.43, 0.90, 0.68 4f107cab5879de2e9a67f2bdd0d926cad2d0d262
c881d78 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:29:49 up 4:15, 2 users, load average: 0.62, 0.99, 0.67 6eb6c61590fd820c93539809e4e8de6f6c486755
42b8f74 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:24:54 up 4:10, 2 users, load average: 0.37, 0.40, 0.37 926e23dbf809bd54e3fafd22bc4cf4250be0f82
0383ce4 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:23:35 up 4:09, 2 users, load average: 0.35, 0.39, 0.36 34cc1117664a114fdaa2c27721798b12711a315c
14c7721 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:23:16 up 4:08, 2 users, load average: 0.41, 0.40, 0.37 32ab8a1ea0a6f0032906a25de63f041d81ea2a71
5f4fb3 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:21:55 up 4:07, 2 users, load average: 0.42, 0.34, 0.35 a7f0f59e2057bea9461dd24f9aa697c59daa239c
3255e1e > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:20:08 up 4:05, 2 users, load average: 0.15, 0.26, 0.32 60b9ab52682430542bed4361b949deeacc41e27a
e5d69f0 > run 162020130 shaozhenzhe Linux Debian 4.19.0-18-686 #1 SMP Debian 4.19.208-1 (2021-09-29) i686 GNU/Linux 19:17:13 up 4:02, 2 users, load average: 0.10, 0.26, 0.33 26ad44421a16982299b3ad0ef19c2e9fe858bb31
```

git branch 截图

```
shaozhenzhe@Debian:~/ics2022$ git branch
  master
   pa0
   pa1
   pa2
*  pa3
shaozhenzhe@Debian:~/ics2022$
```

实验内容

实现 loader (10分)

- 实现简单 loader，触发未实现指令 int；

在 nanos-lite/src/main.c 中定义宏 HAS_ASYE

根据讲义用到 ramdisk_read 函数，其中第一个参数填入 DEFAULT_ENTRY，偏移量为 0，长度为 ramdisk 的大小即可。

记得声明外部函数。

nanos-lite/src/loader.c

```
extern void ramdisk_read(const void *buf, off_t offset, size_t len);
extern size_t get_ramdisk_size();

uintptr_t loader(_Protect *as, const char *filename) {
    //TODO();
    ramdisk_read(DEFAULT_ENTRY, 0, get_ramdisk_size());
    return (uintptr_t)DEFAULT_ENTRY;
}
```

运行结果

```
shaozhenzhe@Debian: ~/ics2022/nanos-lite
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:55:03, May 27 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 17:11:02, May 27 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101a60, end = 0x106d9c,
size = 21308 bytes
invalid opcode(eip = 0x04001ec0): cd 80 5b 5d c3 66 90 90 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x04001ec0 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x04001ec0) in the disassembling result to distinguish which case
it is.

If it is the first case, see
0x04001ec0: cd 80 5b 5d c3 66 90 90
for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

(nemu) █
```

- 实现引入文件系统后的 loader;

这一步是在运行测试用例 `/bin/text` 时实现的

先在 `nanos-lite/Makefile` 修改如下

```
update: update-ramdisk-fsimg src/syscall.h
@touch src/initrd.S
```

然后在loader函数改为从文件读取

```
uintptr_t loader(_Protect *as, const char *filename) {
    //TODO();
    //读取文件位置
    int index=fs_open(filename,0,0);
    //读取长度
    size_t length=fs_filesz(index);
    //读取内容
    fs_read(index,DEFAULT_ENTRY,length);
    //关闭文件
    fs_close(index);
    //ramdisk_read(DEFAULT_ENTRY, 0, get_ramdisk_size());
    return (uintptr_t)DEFAULT_ENTRY;
}
```

添加寄存器和 LIDT 指令 (10 分)

- 根据 i386 手册正确添加 IDTR 和 CS 寄存器;

`nemu/include/cpu/reg.h`, idtr和CS寄存器实现如下

```
typedef struct {
    union{
        union{
            uint32_t _32;
            uint16_t _16;
        }
    }
};
```

```

        uint8_t _8[2];
    }gpr[8];

    /* Do NOT change the order of the GPRs' definitions. */

    /* In NEMU, rtlreg_t is exactly uint32_t. This makes RTL instructions
     * in PA2 able to directly access these registers.
     */
    struct{
        rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
};
vaddr_t eip;

union{
    uint32_t val;
    struct{
        uint32_t CF:1;
        uint32_t :5; //无名位域
        uint32_t ZF:1;
        uint32_t SF:1;
        uint32_t :1;
        uint32_t IF:1;
        uint32_t :1;
        uint32_t OF:1;
    };
}eflags;

struct {
    uint32_t base; //32位base
    uint16_t limit; //16位limit
}idtr;
uint32_t cs;

} CPU_state;

```

- 在 restart() 中正确设置寄存器初始值;

nemu/src/monitor/monitor.c

根据讲义在 restart() 函数中将 EFLAGS 初始化为 0x2

```

static inline void restart() {
    /* Set the initial instruction pointer. */
    cpu.eip = ENTRY_START;
    cpu.eflags.val=0x2;
    cpu.cs=0x8;

#ifdef DIFF_TEST
    init_qemu_reg();
#endif
}

```

- LIDT 指令细节可在 i386 手册中找到;

nemu/src/cpu/exec/exec.c

```
make_group(gp7,
    EMPTY, EMPTY, EMPTY, EX(lidt),
    EMPTY, EMPTY, EMPTY, EMPTY)
```

根据手册, 如果 OperandSize 是 16, 则 limit 读取 16 位, base 读取 24 位。如果 OperandSize 是 32, 则 limit 读取 16 位, base 读取 32 位

nemu/src/cpu/exec/system.c

```
make_EHelper(lidt) {
    //TODO();
    cpu.idtr.limit=vaddr_read(id_dest->addr,2);//limit16
    if (decoding.is_operand_size_16){
        cpu.idtr.base=vaddr_read(id_dest->addr+2,3);//base24
    }
    else{
        cpu.idtr.base=vaddr_read(id_dest->addr+2,4);//base32
    }

    print_asm_template1(lidt);
}
```

实现 INT 指令 (10 分)

- 实现写在 raise_intr() 函数中;

nemu/src/cpu/intr.c

根据手册完成 raise_instr() 函数

```
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
    /* TODO: Trigger an interrupt/exception with ``NO''.
     * That is, use ``NO'' to index the IDT.
     */

    //TODO();
    //获取门描述符
    vaddr_t gate_addr=cpu.idtr.base+8*NO;
    //P位校验
    if (cpu.idtr.limit<0){
        assert(0);
    }
    //将eflags、cs、返回地址压栈
    rtl_push(&cpu.eflags.val);
    rtl_push(&cpu.cs);
    rtl_push(&ret_addr);
    //组合中断处理程序入口点
    uint32_t high,low;
    low=vaddr_read(gate_addr,4)&0xffff;
    high=vaddr_read(gate_addr+4,4)&0xffff0000;
    //设置eip跳转
    decoding.jump_eip=high|low;
    decoding.is_jump=true;
}
```


- 使用 INT 的 helper 函数调用 raise_intr();

在 int 指令的 helper 函数中调用 raise_intr() 即可, 要声明 raise_intr 外部函数

nemu/src/cpu/exec/system.c

```
extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
make_EHelper(int) {
    //TODO();
    raise_intr(id_dest->val, decoding.seq_eip);

    print_asm("int %s", id_dest->str);

#ifdef DIFF_TEST
    diff_test_skip_nemu();
#endif
}
```

最后填表

```
/* 0xcc */    EMPTY, IDEXW(I, int, 1), EMPTY, EMPTY,
```

- 指令细节可在 i386 手册中找到;

运行结果

```
shaozhenzhe@Debian: ~/ics2022/nanos-lite
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101d80, end = 0x1070bc, size = 2
1308 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
invalid opcode(eip = 0x0010128f): 60 54 e8 15 fd ff ff 83 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010128f is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010128f) in the disassembling result to distinguish which case it is.

If it is the first case, see
0x0010128f: int 0x0010128f
for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

[src/monitor/diff-test/diff-test.c,164,difftest_step] reg diff NEMU=0x00007b40 QEMU=0x00
007b20
[src/monitor/diff-test/diff-test.c,167,difftest_step] reg diff NEMU=0x00101297 QEMU=0x00
101290
(nemu)
```

实现其他相关指令和结构体 (15 分)

- 组织 _RegSet 结构体, 需要说明理由;

根据讲义, 现场保存的顺序为: 1. 硬件保存 EFLAGS, CS, EIP 2. vecsys() 压入错误码和异常号

#irq 3. asm_trap() 把用户进程的通用寄存器保存

寄存器保存的顺序可以看手册的 pusha 顺序。

```

IF OperandSize = 16 (* PUSHA instruction *)
THEN
    Temp ← (SP);
    Push(AX);
    Push(CX);
    Push(DX);
    Push(BX);
    Push(Temp);
    Push(BP);
    Push(SI);
    Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
    Temp ← (ESP);
    Push(EAX);
    Push(ECX);
    Push(EDX);
    Push(EBX);
    Push(Temp);
    Push(EBP);
    Push(ESI);
    Push(EDI);
FI;

```

根据这些顺序倒序恢复，因此 `_RegSet` 结构体如下

nexus-am/am/arch/x86-nemu/include/arch.h

```

struct _RegSet {
    uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
    int      irq;
    uintptr_t error_code, eip, cs, eflags;
};

```

- pusha, popa, iret;

pusha:

nemu/src/cpu/exec/data-mov.c, 根据上面的图按顺序push即可

```

make_EHelper(pusha) {
    //TODO();
    t0 = cpu.esp;
    rtl_push(&cpu.eax);
    rtl_push(&cpu.ecx);
    rtl_push(&cpu.edx);
    rtl_push(&cpu.ebx);
    rtl_push(&t0);
    rtl_push(&cpu.ebp);
    rtl_push(&cpu.esi);
    rtl_push(&cpu.edi);

    print_asm("pusha");
}

```

填表

```
/* 0x60 */ EX(pusha), EMPTY, EMPTY, EMPTY,
```

popa:

Operation

```
IF OperandSize = 16 (* instruction = POPA *)
THEN
    DI ← Pop();
    SI ← Pop();
    BP ← Pop();
    throwaway ← Pop (); (* Skip SP *)
    BX ← Pop();
    DX ← Pop();
    CX ← Pop();
    AX ← Pop();
ELSE (* OperandSize = 32, instruction = POPAD *)
    EDI ← Pop();
    ESI ← Pop();
    EBP ← Pop();
    throwaway ← Pop (); (* Skip ESP *)
    EBX ← Pop();
    EDX ← Pop();
    ECX ← Pop();
    EAX ← Pop();
FI;
```

按照手册，按顺序pop即可

```
make_EHelper(popa) {
    //TODO();
    rtl_pop(&cpu.edi);
    rtl_pop(&cpu.esi);
    rtl_pop(&cpu.ebp);
    rtl_pop(&t0);
    rtl_pop(&cpu.ebx);
    rtl_pop(&cpu.edx);
    rtl_pop(&cpu.ecx);
    rtl_pop(&cpu.eax);

    print_asm("popa");
}
```

填表

```
/* 0x60 */    EX(pusha), EX(popa), EMPTY, EMPTY,
```

iret:

按顺序 `eip cs eflags` 出栈即可

```
make_EHelper(iret) {
    //TODO();
    rtl_pop(&decoding.jump_eip);
    decoding.is_jump=1;
    rtl_pop(&cpu.cs);
    rtl_pop(&cpu.eflags.val);

    print_asm("iret");
}
```

填表

```
/* 0xcc */    EMPTY, IDEXW(I,int,1), EMPTY, EX(iret),
```

- 上述指令细节都可在 i386 手册中找到;

运行结果: 触发了ID=8的 BAD TRAP

```
shaozhenzhe@Debian: ~/ics2022/nanos-lite
+ CC arch/x86-nemu/src/ioe.c
+ AR /home/shaozhenzhe/ics2022/nexus-am/am/build/am-x86-nemu.a
make[2]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/am'
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am'
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make: [/home/shaozhenzhe/ics2022/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nemu'
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/exec/system.c
+ LD build/nemu
./build/nemu -l /home/shaozhenzhe/ics2022/nanos-lite/build/nemu-log.txt /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:07:40, May 27 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 19:41:03, May 27 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101d80, end = 0x1070bc, size = 21308 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/irq.c,5,do_event] system panic: Unhandled event ID = 8
nemu: HIT BAD TRAP at eip = 0x001000f1
(nemu)
```

完善事件分发和 do_syscall (15 分)

- 完善 do_event, 目前阶段仅需要识别出系统调用事件即可;

识别系统调用事件 `_EVENT_SYSCALL`, 然后调用 `do_syscall()` 即可, 同时要声明函数 `do_syscall()`

`nanos-lite/src/irq.c`

```
extern _RegSet* do_syscall(_RegSet *r);

static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            do_syscall(r);
            break;
        default: panic("Unhandled event ID = %d", e.event);
    }

    return NULL;
}
```

- 添加整个阶段中的所有系统调用 (none, exit, brk, open, write, read, lseek, close) ;

首先在 `nexus-am/am/arch/x86-nemu/include/arch.h` 中实现正确的 `SYSCALL_ARGx()` 宏

```
#define SYSCALL_ARG1(r) r->eax
#define SYSCALL_ARG2(r) r->ebx
#define SYSCALL_ARG3(r) r->ecx
#define SYSCALL_ARG4(r) r->edx
```

完善 `do_syscall()` 函数, `nanos-lite/src/syscall.c`

```

_RegSet* do_syscall(_RegSet *r) {
    uintptr_t a[4];
    a[0] = SYSCALL_ARG1(r);
    a[1] = SYSCALL_ARG2(r);
    a[2] = SYSCALL_ARG3(r);
    a[3] = SYSCALL_ARG4(r);
    switch (a[0]) {
        default: panic("Unhandled syscall ID = %d", a[0]);
    }

    return NULL;
}

```

添加目前的系统调用 `SYS_none` 和 `SYS_exit`

`SYS_none`，根据讲义，这个系统调用什么都不用做，直接返回 1 即可

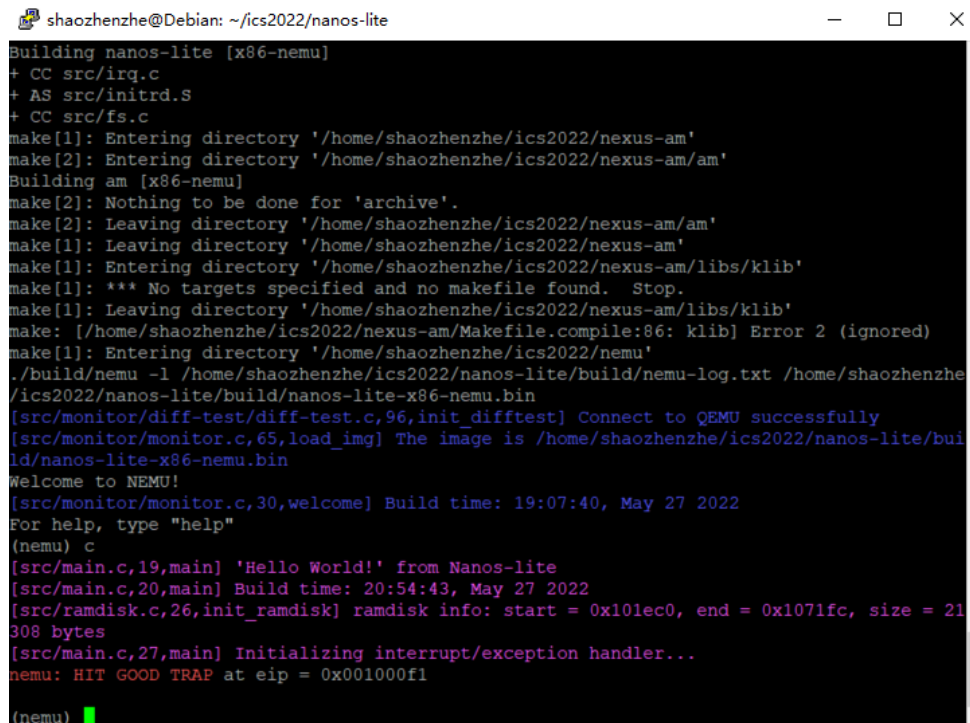
`SYS_exit` 接收一个退出状态的参数，用这个参数调用 `_halt()` 即可，这个参数尝试后发现有 `SYSCALL_ARG2(r)`

```

case SYS_none:
    SYSCALL_ARG1(r)=1;
    break;
case SYS_exit:
    _halt(SYSCALL_ARG2(r));
    break;

```

运行结果



```

shaozhenzhe@Debian: ~/ics2022/nanos-lite
Building nanos-lite [x86-nemu]
+ CC src/irq.c
+ AS src/initrd.S
+ CC src/fs.c
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am'
make[2]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/am'
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am'
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make: [/home/shaozhenzhe/ics2022/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nemu'
./build/nemu -l /home/shaozhenzhe/ics2022/nanos-lite/build/nemu-log.txt /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:07:40, May 27 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 20:54:43, May 27 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101ec0, end = 0x1071fc, size = 21308 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x001000f1
(nemu)

```

还有其他的系统调用在后面的过程中逐渐补完，这里放一个最终的 `do_syscall()` 函数

```

_RegSet* do_syscall(_RegSet *r) {
    uintptr_t a[4];
    a[0] = SYSCALL_ARG1(r);
    a[1] = SYSCALL_ARG2(r);
    a[2] = SYSCALL_ARG3(r);

```

```

a[3] = SYSCALL_ARG4(r);
switch (a[0]) {
    case SYS_none:
        SYSCALL_ARG1(r)=1;
        break;
    case SYS_exit:
        _halt(SYSCALL_ARG2(r));
        break;
    case SYS_brk:
        SYSCALL_ARG1(r)=0;
        break;
    case SYS_open:
        SYSCALL_ARG1(r)=(int)sys_open(a[1],a[2],a[3]);
        break;
    case SYS_write:
        SYSCALL_ARG1(r)=(int)sys_write(a[1],a[2],a[3]);
        break;
    case SYS_read:
        SYSCALL_ARG1(r)=(int)sys_read(a[1],a[2],a[3]);
        break;
    case SYS_lseek:
        SYSCALL_ARG1(r) = (int)sys_lseek(a[1], a[2], a[3]);
        break;
    case SYS_close:
        SYSCALL_ARG1(r)=(int)sys_close(a[1]);
        break;
    default: panic("Unhandled syscall ID = %d", a[0]);
}

return NULL;
}

```

实现堆区管理 (10 分)

- 堆区管理相关系统调用和封装函数，如 `sys_brk`，`mm_brk`，`_sbrk()`，`brk()`，`sbrk()`；
- 根据讲义，让 `sys_brk` 系统调用总是返回 0 即可

```

case SYS_brk:
    SYSCALL_ARG1(r)=0;
    break;

```

`_sbrk()` 实现，根据讲义实现即可

1. program break 一开始的位置位于 `_end`
2. 被调用时，根据记录的 program break 位置和参数 `increment`，计算出新 program break
3. 通过 `sys_brk` 系统调用来让操作系统设置新 program break
4. 若 `sys_brk` 系统调用成功，该系统调用会返回 0，此时更新之前记录的 program break 的位置，并将旧 program break 的位置作为 `_sbrk()` 的返回值返回
5. 若该系统调用失败，`_sbrk()` 会返回 -1

`navy-apps/libs/libos/src/nanos.c`

```

extern char _end; //声明外部变量
static intptr_t brk=(intptr_t)&_end; //记录开始位置
void *_sbrk(intptr_t increment){
    intptr_t old_brk = brk;

```

```

    intptr_t new_brk=old_brk+increment;//记录增加后的位置
    intptr_t res = _syscall_(SYS_brk,new_brk,0,0);//系统调用
    if(res == 0){    //若成功，返回旧 program break 的位置
        brk= new_brk;
        return (void*)old_brk;
    }
    else{
        return (void*) -1;
    }
}

```

- 其中有部分已经实现，或者只给出了框架；

实现系统调用（10 分）

- 实现 sys_[open|write|read|lseek|close|brk] 函数；

sys_open()

调用 fs_open ，根据给定路径、标志和打开模式打开文件，注意pathname类型转换

nanos-lite/src/syscall.c

```

static inline uintptr_t sys_open(uintptr_t pathname, uintptr_t flags,
    uintptr_t mode) {
    //TODO();
    return fs_open((char *)pathname,flags,mode);;
}

```

在 navy-apps/libs/libos/src/nanos.c 的 _open() 中修改对应的接口函数

```

int _open(const char *path, int flags, mode_t mode) {
    //_exit(SYS_open);
    _syscall_(SYS_open, (uintptr_t)path, flags, mode);
}

```

sys_write()

和 sys_open 类似，直接调用 fs_write

```

static inline uintptr_t sys_write(uintptr_t fd, uintptr_t buf, uintptr_t
    len) {
    //TODO();
    return fs_write(fd, (void *)buf, len);
}

```

在 navy-apps/libs/libos/src/nanos.c 的 _write() 中调用系统调用接口函数

```

int _write(int fd, void *buf, size_t count){
    //_exit(SYS_write);
    _syscall_(SYS_write, fd, (uintptr_t)buf, count);
}

```

sys_read()

```
static inline uintptr_t sys_read(uintptr_t fd, uintptr_t buf, uintptr_t len)
{
    //TODO();
    return fs_read(fd, (void *)buf, len);
}
```

系统调用接口函数

```
int _read(int fd, void *buf, size_t count) {
    //_exit(SYS_read);
    _syscall_(SYS_read, fd, (uintptr_t)buf, count);
}
```

sys_lseek()

框架已实现，只需要在 do_syscall() 添加调用

sys_close()

```
static inline uintptr_t sys_close(uintptr_t fd) {
    //TODO();
    return fs_close(fd);
}
```

系统调用接口函数

```
int _close(int fd) {
    //_exit(SYS_close);
    _syscall_(SYS_close, fd, 0, 0);
}
```

sys_brk()

之前已实现

- 注意，文件系统相关接口已经实现，你只需要自行确定相关参数调用即可；
相对应的 do_syscall() 添加调用已经在上面给出

成功运行各测试用例（20 分）

- Hello world;


```
shaozhenzhe@Debian: ~/ics2022/nanos-lite
make[2]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/am'
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am'
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make: [/home/shaozhenzhe/ics2022/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nemu'
+ CC src/cpu/exec/exec.c
+ LD build/nemu
./build/nemu -l /home/shaozhenzhe/ics2022/nanos-lite/build/nemu-log.txt /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,96,init_diffptest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:07:40, May 27 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:10:18, May 27 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101ec0, end = 0x1081fc, size = 25404 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
Hello World!
Hello World for the 2th time
Hello World for the 3th time
Hello World for the 4th time
Hello World for the 5th time
Hello World for the 6th time
Hello World for the 7th time
Hello World for the 8th time
```

- /bin/text;

nanos-lite/src/main.c

```
uint32_t entry = loader(NULL, "/bin/text");
```

```
shaozhenzhe@Debian: ~/ics2022/nanos-lite
make[2]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/am'
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am'
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/shaozhenzhe/ics2022/nexus-am/libs/klib'
make: [/home/shaozhenzhe/ics2022/nexus-am/Makefile.compile:86: klib] Error 2 (ignored)
make[1]: Entering directory '/home/shaozhenzhe/ics2022/nemu'
./build/nemu -l /home/shaozhenzhe/ics2022/nanos-lite/build/nemu-log.txt /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,96,init_diffptest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/shaozhenzhe/ics2022/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:57:28, May 28 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 19:00:03, May 28 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102260, end = 0x43906e6, size = 69788806 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x001000f1
(nemu) █
```

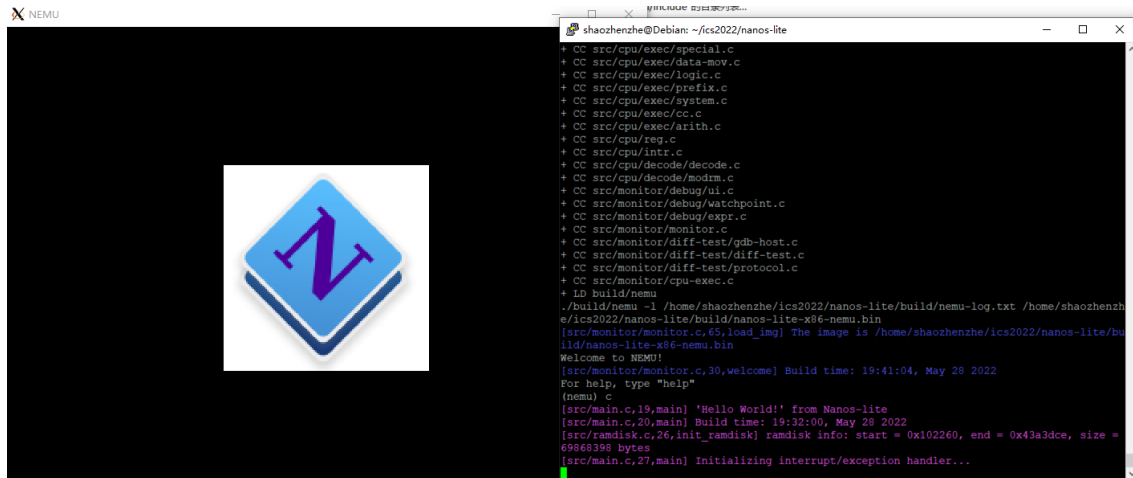
- /bin/bmptest;

nanos-lite/src/main.c

```
uint32_t entry = loader(NULL, "/bin/bmptest");
```

navy-apps/Makefile.compile, 修改o2为o0

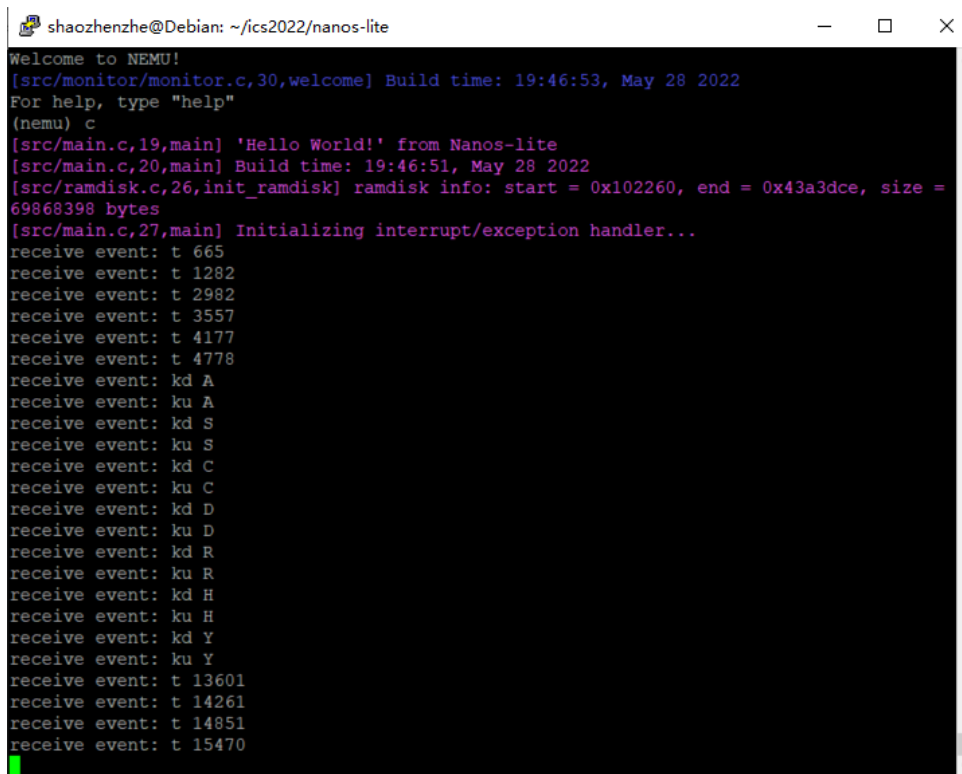
```
CFLAGS += -std=gnu99 -O0 -MMD $(INCLUDES) -D$(ISA_DEF) -fdata-sections -
ffunction-sections -static
CXXFLAGS += -std=c++11 -O0 -MMD $(INCLUDES) -D$(ISA_DEF) -fdata-sections -
ffunction-sections -static
```



- /bin/events;

nanos-lite/src/main.c

```
uint32_t entry = loader(NULL, "/bin/events");
```

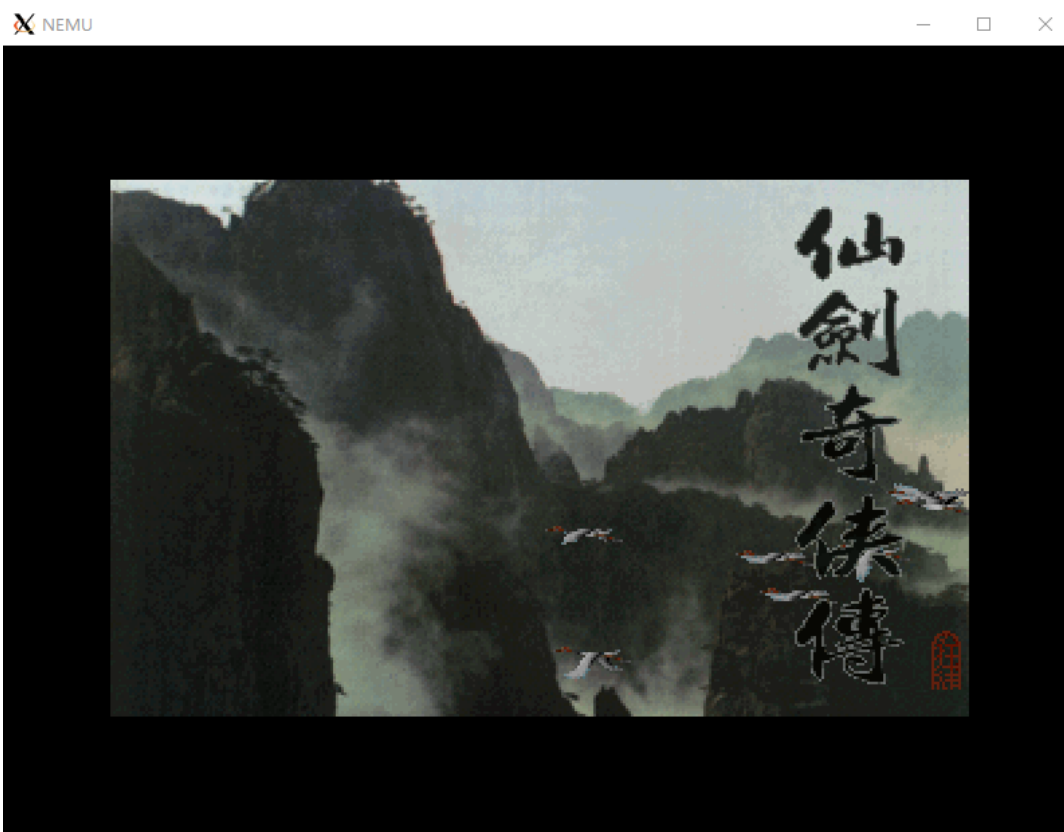


- 仙剑奇侠传;

把仙剑奇侠传文件夹放到放到 `navy-apps/fsimg/share/games/` 目录下

nanos-lite/src/main.c

```
uint32_t entry = loader(NULL, "/bin/pal");
```



遇到的问题及解决办法

1.遇到问题：运行/bin/text时，出现 `system panic : No such file : /share/texts/num` 错误

解决办法：感觉上是编译文件或者ramdisk 镜像文件的问题，干脆全部make clean后重新编译，镜像也重新加载，成功解决。

2.遇到问题：运行/bin/bmptest时很长时间都没有输出。

解决方法：把DEBUG和DIFF-TEST的宏都关了，可以加快执行速度。

实验心得

本次实验实现了一个简单的单任务操作系统，感觉难度挺大的，涉及到了很多计算机底层的知识，对于操作系统的文件系统、异常处理、加载器等等部分有了更深的理解。希望在后面的学习中能够逐渐理解整个操作系统怎么运行的。

其他备注

无