

华中科技大学

2022

系统能力培养 课程实验报告

题 目:	riscv 指令模拟器
专 业:	计算机科学与技术
班 级:	CSXJ1901 班
学 号:	U201911122
姓 名:	闫浩
电 话:	14794520002
邮 件:	U201911122@hust.edu
完成日期:	2022-02-05



目录

1 课程实验概述	3
1.1 课设目的	3
1.2 实验环境	3
2. 实验方案设计	4
2.1 世界诞生前夜---开发环境： 框架。	
2.2 开天辟地---图灵机	
PA1.1 简易调试器	
PA1.2 表达式求值	
PA1.3 监视点与断点	
2.3 简单复杂计算机--冯诺依曼计算机	
PA2.1 运行第一个 C 程序	
PA2.2 丰富指令集，测试所有程序	
PA2.3 实现 I/O 指令，测试打字游戏	
2.4 穿越时空之旅：异常控制流	
PA3.1 实现系统调用	
PA3.2 实现文件系统	
PA3.3 运行仙剑奇侠传	
2.5 问题解答	
3. 实验过程与结果分析	8
4.实验心得与体会	

课程实验概述

1.1 课设目的

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统. 南京大学计算机科学与技术系计算机系统基础课程的小型项目 (Programming Assignment, PA)将提出 x86 架构的一个教学版子集 n86, 指导学生实现一个功能完备的 n86 模拟器 NEMU(NJU EMUlator), 最终在 NEMU 上运行游戏"仙剑奇侠传", 来让学生探究"程序在计算机上运行"的基本原理. NEMU 受到了 [QEMU](#) 的启发, 并去除了大量与课程内容差异较大的部分. PA 包括一个准备实验(配置实验环境)以及 5 部分连贯的实验内容:

- 简易调试器
- 冯诺依曼计算机系统
- 批处理系统
- 分时多任务
- 程序性能优化

1.2 实验环境

- CPU 架构: x64
- 操作系统: Linux(VirtualBox)
- 编译器: GCC
- 编程语言: C 语言

2. 实验方案设计

2.1 PA0

本实验要求完成五个 PA (Programming Assignment), PA 包括一个准备实验以及连贯的实验内容:

- PA0: 开发环境配置(准备实验)
- PA1: 简易调试器
- PA2: 冯诺依曼计算机系统
- PA3: 批处理系统
- PA4: 分时多任务

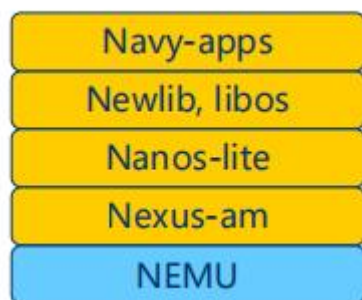
本小节完成 pa0。

PA 指导学生实现一个功能完备的模拟器 NEMU(NJU EMUlator)

- ▶ 支持 x86(教学版子集)/mips32/riscv32
- 最终在 NEMU 上运行真实游戏“仙剑奇侠传”
- 揭示“程序在计算机上运行”的基本原理

PA 组成-NEMU 全系统模拟器

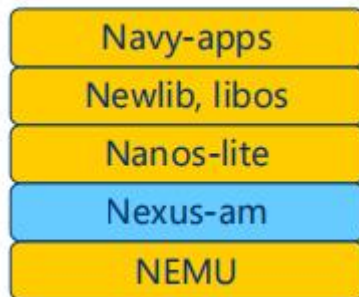
- ▶ 简易调试器(Monitor) - 基础设施
- 单步执行, 打印寄存器/内存, 表达式求值, 监视点
- ▶ CPU
- 完整的指令周期: 取指, 译码, 执行
- 异常处理模块
- 支持分页的 MMU
- ▶ 内存
- ▶ 设备
- 时钟, 键盘, VGA, 串口(功能经过简化)



PA 组成-Nexus-AM ISA 抽象层

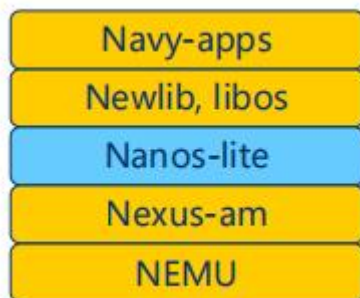
- ▶ 把机器功能抽象成 C 语言 API, 向程序屏蔽 ISA 细节
- ▶ TRM 计算抽象
- _putc() 输出一个字符
- _halt() 终止程序运行
- ▶ IOE 输入输出抽象
- uptime() 返回当前时间
- read_key() 返回按键
- draw_rect() 在屏幕上绘制矩形像素

- ▶ CTE 上下文管理抽象
- 上下文保存/恢复
- 事件处理回调函数
- ▶ VME 虚存抽象
- `_protect()` 创建虚拟地址空间
- `_map()` 添加 `va` 到 `pa` 的映射关系



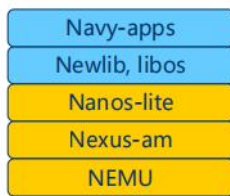
PA 组成-Nanos-lite 简易多任务 OS

- ▶ Nanos(Nanjing U OS)简化版, 能支撑真实程序
- ELF 加载器
- 中断/异常事件分发
- 6+2 个系统调用
- ▶ `open`, `read`, `write`, `lseek`, `close`, `brk`
- ▶ `exit`, `exec`
- 简易 VFS
- ▶ 文件数量, 大小皆固定, 没有目录
- ▶ 文件系统基于 `ramdisk`, 无需持久化
- ▶ 3 个设备文件(映射到设备抽象层)
- `/dev/fb`, `/dev/events`, `/proc/dispinfo`
- 设备抽象层
- 简易分页存储管理 (顺序分配, 不释放)
- 两个进程的简易轮转调度



库和应用程序 Navy-apps

- ▶ Newlib – C 库, 支持 POSIX 标准
- ▶ libos – 系统调用接口
- ▶ Navy-apps – 应用程序集
- `hello`, 仙剑奇侠传, LiteNES (PA 用)
- Lua 解释器, `busybox` 套件(`shell`, `vi`)
- NWM 窗口管理器, Nterm 终端
- NCC 编译器



2.2 PA1

PA1 最终要求实现一个简易调试器，相当于一个简化版的 `gdb`，要求实现单步执行、打印寄存器状态、扫描内存、表达式求值、监视点等功能。

2.2.1 PA1.1

需要实现单步执行，打印寄存器状态，扫描内存，需要在 `ui.c` 中声明并实现相关的函数。三个功能需要实现：单步执行，打印寄存器信息，扫描内存。在 `ui.c` 中，这三个功能分别用 `cmd_si`、`cmd_info` 和 `cmd_x` 表示。`cmd_si` 通过调用 `cpu_exec` 完成指定步数的执行，`cmd_info` 调用 `isa_reg_display` 打印寄存器名称和值，`isa_reg_display` 使用 `reg_name` 和 `reg_l` 打印寄存器的信息，`cmd_x` 对输入的参数进行分析，通过 `vaddr_read` 函数读取对应地址的数值并输出。

2.2.2 PA1.2

进行表达式求值，首先要在 `expr.c` 中补充规则和相应的正则表达式，例如 十六进制数的正则表达式与 `token` 类型为 `{"0[Xx][0-9a-fA-F]+", TK_HEX}`。

随后完成函数 `make_token`，用于为算术表达式中的各种 `token` 类型添加规则，并在成功识别出 `token` 后，将 `token` 的信息依次记录到 `tokens` 数组中。接着完成括号匹配函数 `check_parentheses`，完成运算符优先级函数 `op_priority`，根据 C 语言中的优先级来实现对应的操作符的优先级。

2.2.3 PA1.3

完善 `watchpoint` 结构，随后在 `watchpoint.c` 中实现函数 `new_wp`，

`free_wp`，`print_wp`。在 `ui.c` 中使用上述函数完成命令 `cmd_x` 与 `cmd_d`，

完善函数 `cmd_info`，同时要在 `cpu.exec` 中实现当监视点状态改变使程序暂停执行的逻辑，该逻辑较为简单，程序每执行一步就对所有的监视点进行表达式求值，并与旧值进行比较，如果两值不同，则置 `nemu` 的状态为 `NEMU_STOP`。

2.3 PA2

2.3.1 PA2.1

实现对于部分指令的支持，需要先运行 `dummy.c` 并在反汇编代码中找出未实现的指令，随后通过查询手册确定需要实现的新指令，并在 `all-instr.h` 中定义相关的执行函数，修改 `decode.c`、`rtl.h`、`compute.c`、`control.c` 等文件实现相应的译码和执行辅助函数，最后重新编译运行即可。

2.3.2 PA2.2

要求在 `NEMU` 中实现更多的指令，这需要先找出反汇编代码中的未实现的指令，再编写译码和执行函数。先完成 `diff-test` 会方便测试。实现指令的过程类似于 PA2.1，以 `sum.c` 为例，需要实现的指令包括 `beq`、`bne`、`add` 和 `sltiu`。译码函数在对应的辅助函数中编写，执行函数在 `control.c` 和 `compute.c` 中编写，要根据指令的相关行为编写对应的执行辅助函数

2.3.3 PA2.3

在相关设备文件中完成四个输入输出设备程序的编写，包括串口，时钟，键盘和 VGA。串口已经在 `trm.c` 中实现，时钟需要在 `nemu-timer.c` 中完善，键盘需要在 `nemu-input.c` 中完善，VGA 需要在 `nemu-video.c` 中完善。

2.4 PA3

主要任务是实现系统调用和文件系统，使得 nemu 最终能够运行仙剑奇侠传

2.4.1 PA3.1

现自陷操作 `_yield` 及其过程，要实现自陷操作，首先要在 `cpu`

结构中添加控制状态寄存器，然后实现自陷需要的指令，通过异常号识别出自陷异常，完成自陷事件。实现自陷指令，需要实现的指令有 `csrrs`、`csrrw`、`ecall` 和 `sret`，通过 `ecall` 指令进入自陷操作，`csrrs` 和 `csrrw` 指令对控制状态寄存器进行修改，`sret` 指令用于自陷后返回，然后需要在 `intr.c` 文件中进行 `raise_intr` 函数的编写，`raise_intr` 函数用于模拟相应过将当前 PC 值保存到 `sepc` 寄存器，在 `scause` 寄存器中设置异常号，从 `stvec` 寄存器中取出异常入口地址，跳转到异常入口地址；触发自陷操作后，需要保存上下文，根据 `trap.S` 汇编代码的压栈的顺序重构 `Context` 成员体结构，接着要实现正确的事件分发，需要在 `_am_irq_handle` 函数中通过异常号识别出自陷异常，在 `do_event` 函数中识别出自陷事件 `_EVENT_YIELD`，最后通过 `sret` 指令返回并恢复上下文。

2.4.2 PA3.2

需要实现用户程序的加载和系统调用，支撑 TRM 程序的运行，需要通过实现 `loader` 函数，增加系统调用，完善堆区管理函数。实现用户程序的加载和系统调用，支撑 TRM 程序的运行，为了加载用户程序首先要实现 `loader` 函数，因为目前还没有实现文件系统，所以直接在 `loader` 函数中通过 `ramdisk_read` 函数把可执行文件中的代码和数据放置在正确的内存位置，然后跳转到程序入口，接着需要完成系统调用的实现，和 3.1 中识别出自陷事件类似，让 `nemu` 能够识别出系统调用，然后在 `do_event` 中添加 `do_syscall` 的调用，根据 `nanos.c` 中的 `ARGS_ARRAY` 在 `riscv32-nemu` 中实现正确的 GPR 宏，随后添加 `SYS_yield`、`SYS_read`、`SYS_write` 和 `SYS_brk` 系统调用，完成函数 `_sbrk` 实现堆区管理。

2.4.3 PA3.3

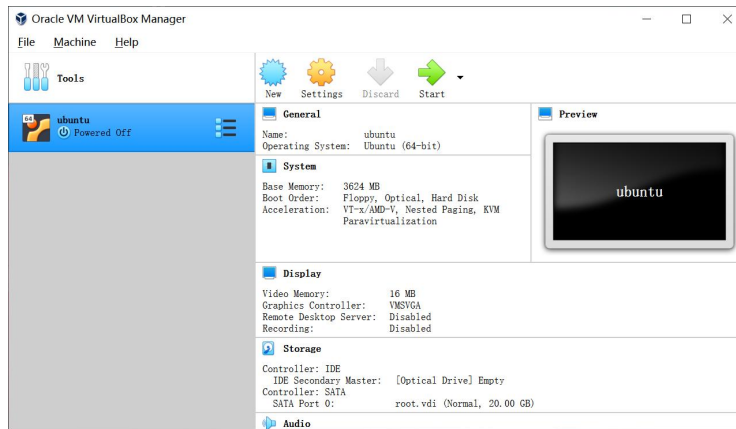
需要实现文件系统，完成 `fs_open`、`fs_read` 等函数，完成设备函数的编写，

使用 `fs` 函数修改 `loader`，最终可运行仙剑奇侠传。需要实现文件系统，添加设备的支持，最终能够运行仙剑奇侠传。首先需要实现函数 `fs_open`，`fs_read` 和 `fs_close`，因为 `ramdisk` 中的文件数量增加之后，就不适合直接在 `loader` 函数中直接使用 `ramdisk_read`，在完成了这几个 `fs` 函数后，替换掉 `loader` 函数中的 `ramdisk_read` 函数，修改其逻辑这样就可以在 `loader` 中使用文件名来指定加载的程序了，随后完成 `fs_write` 和 `fs_lseek` 函数，使其能够进行输出，完成这些函数后，要补充相关的系统调用；要实现虚拟文件系统 VFS，把 IOE 抽象成文件，首先需要在 VFS 中补充多种特殊文件的支持，接着实现函数 `serial_write`，完成串口的写入，然后完成实现 `events_read` 函数，支持读操作，最后需要完成 `init_fs`、`fb_write`、`fb_sync_write`、`init_device`、`dispinfo_read` 函数，实现对 VGA 设备的支持。要注意的是因为在 `Finfo` 结构中添加了读函数指针和写函数指针，所以要修改 `fs_write` 和 `fs_read` 的逻辑，完成这些以后，如果没有错误，就能够运行仙剑奇侠传了。

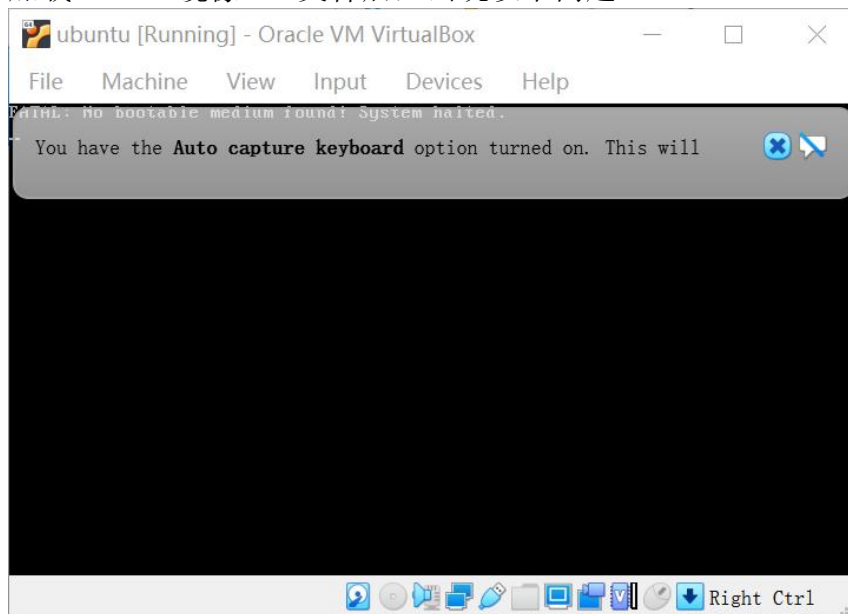
3. 实验过程与结果分析

3.1 PA0:世界诞生的前夜：开发环境配置

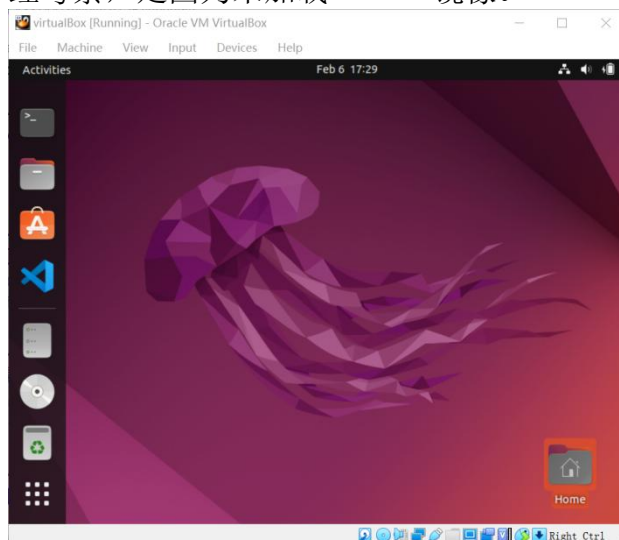
首先是进行开发环境的配置,本次实验使用 virtualbox 虚拟机搭建 ubuntu 系统进行实验,下面是 virtualbox 的环境配置。



加载 ubuntu 镜像 vdi 文件后, 出现以下问题。



经考察, 是因为未加载 ubuntu 镜像。



VirtualBox [Running] - Oracle VM VirtualBox

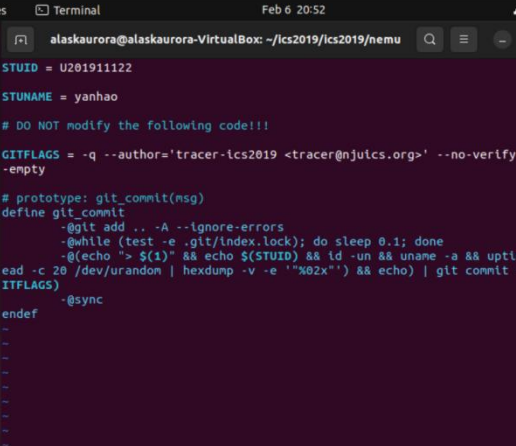
File Machine View Input Devices Help

Activities Terminal Feb 6 20:13

alaskaurora@alaskaurora-VirtualBox: ~/ics2019/ics2019

```
alaskaurora@alaskaurora-VirtualBox:~$ ls
Desktop  Downloads  Music  Public  Templates
Documents  ics2019  Pictures  snap  Videos
alaskaurora@alaskaurora-VirtualBox:~$ cd ics2019
alaskaurora@alaskaurora-VirtualBox:~/ics2019$ ls
key  key.txt
alaskaurora@alaskaurora-VirtualBox:~/ics2019$ ls
alaskaurora@alaskaurora-VirtualBox:~/ics2019$ git init
Reinitialized existing Git repository in /home/alaskaurora/ics2019/.git/
alaskaurora@alaskaurora-VirtualBox:~/ics2019$ git clone https://github.com/supr
emehog/ics2019
Cloning into 'ics2019'...
remote: Enumerating objects: 1640, done.
remote: Counting objects: 100% (1640/1640), done.
remote: Compressing objects: 100% (1304/1304), done.
remote: Total 1640 (delta 268), reused 1640 (delta 268), pack-reused 0
Receiving objects: 100% (1640/1640), 2.83 MiB | 1.77 MiB/s, done.
Resolving deltas: 100% (268/268), done.
alaskaurora@alaskaurora-VirtualBox:~/ics2019$ ls
ics2019
alaskaurora@alaskaurora-VirtualBox:~/ics2019$ cd ics2019
alaskaurora@alaskaurora-VirtualBox:~/ics2019/ics2019$ ls
init.sh  Makefile  nanos-lite  navy-apps  menu  nexus-am  README.md
alaskaurora@alaskaurora-VirtualBox:~/ics2019/ics2019$
```

Right Ctrl



VirtualBox [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Activities Terminal Feb 6 20:52

alaskaurora@alaskaurora-VirtualBox: ~/lcs2019/lcs2019/nemu

```
STUID = U201911122
STUNAME = yanhao

# DO NOT modify the following code!!!

GITFLAGS = -q --author='tracer-lcs2019 <tracer@njucls.org>' --no-verify --allow
-empty

# prototype: git_commit(msg)
define git_commit
    -@git add .. -A --ignore-errors
    -@while (test -e .git/index.lock); do sleep 0.1; done
    -@echo "> $(1)" && echo $(STUID) && id -un && uptime && (h
ead -c 20 /dev/urandom | hexdump -v -e "%02x") && echo) | git commit -F - $(G
ITFLAGS)
    -@sync
endef

"Makefile.git" 15L, 473B

2,0-1 All
```

VirtualBox [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Activities Terminal Feb 6 20:59

alaskaurora@alaskaurora-VirtualBox: ~/lcs2019/lcs2019/nemu

```

Date:      Fri Apr 1 12:13:56 2022 +0800

added omitted logo.txt

commit 9b744ee62c19c102ff12f806104ee69b30f89d5
Author: Jle <jle@opstor.cn>
Date:      Sun Sep 5 21:35:10 2021 +0800

modified for lcs-rvc

commit c9725b7dd55642231978b25e4a935eb603dbf7a
Author: Jle <jle@uncu.lo>
Date:      Tue Sep 8 11:05:54 2020 +0800

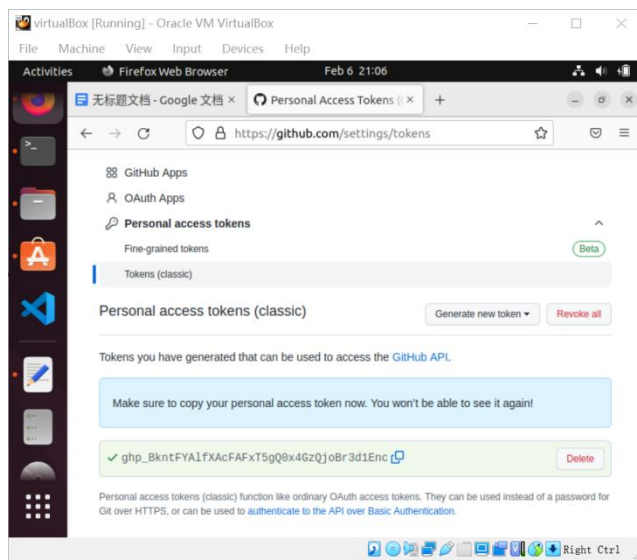
removed calling setup.sh from init.sh

commit 3a944e8f7bb069d3d5e2d2a144c4f7db55c3aa
Author: Jle <jle@uncu.lo>
Date:      Tue Sep 8 10:00:35 2020 +0800

alaskaurora@alaskaurora-VirtualBox:~/lcs2019/nemu$ git push
Username for 'https://github.com': supremeghost
Password for 'https://supremeghost@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently
recommended modes of authentication.
fatal: Authentication failed for 'https://github.com:supremeghost/lcs2019/'
alaskaurora@alaskaurora-VirtualBox:~/lcs2019/nemu$
  
```

Right Ctrl

2



Github 更改 token，之后顺利更改 git 中学号姓名文件

至此，pa0 开发环境配置结束。

3.2PA1:

寄存器

```
union{
    struct {
        rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
    union{
        paddr_t _32;
        ioaddr_t _16;
        bool _8[2];
    } gpr[8];
};
```

使用联合来公用 32 位寄存器的低 16 位和 16 位寄存器和两个 8 位寄存器。

2) 单步执行

首先判断输入参数是否为空， 空则单步执行 1， 不为空则判断参数是否 为数字， 为数字则执行 n 条指令， 不为数字则输出报错信息。

3) 打印寄存器状态

```
static void printRegister(int size, int index){
    switch(size) {
        case 1: printf("%s\t%04X\n", reg_name(index, 1), reg_b(index)); break;
        case 2: printf("%s\t%06X\n", reg_name(index, 2), reg_w(index)); break;
        case 4: printf("%s\t%010X\n", reg_name(index, 4), reg_l(index)); break;
        case -1: printf("eip\t%010X\n", cpu.eip); break;
        default: break;
    }
}
```

完成打印单个寄存器信息的函数， 之后遍历调用即可。

4) 扫描内存

```
static int cmd_x(char *args) //scan memory
{
    if(args == NULL){
        printf("Nothing to scan!\n");
        return 0;
    }

    char *Size = strtok(args, " ");
    int size = -1;
    sscanf(Size, "%d", &size);
    if(size < 0){
        printf("Wrong size to scan!\n");
        return 0;
    }

    char *Addr = Size + strlen(Size) + 1;

    char *leftover;
    uint64_t addr = strtoul(Addr, &leftover, 0);
    if(*leftover != '\0'){
        printf("Wrong addr to scan!\n");
        return 0;
    }

    for(int i = 0; i < size; i++){
        printf("%#010X\t\t\n", vaddr_read(addr + i * 4, 4));
    }

    return 0;
}
```

首先读取参数， 读取初始地址和读取的长度， 若不符合标准则输出报错 信息， 之后利用 `vaddr_read()`函数遍历输出内存信息。

5) 算数表达式求值

```
/* " ", TK_NOTYPE, 0); //空格
/* "\\b(0-9)+\\b", TK_NUMBER, 0); //数字
/* "\\b0[xX](0-9-a-fA-F)+\\b", TK_HEX, 0); //十六进制
/* "\\b(eax|ebx|ecx|edx|ebx|ebp|esp|esi|edi|
```

表达式求值过程主要难点在于正则表达式匹配， 填写正则表达式中所需 要匹配的字符， 规则以及算数优先级。并在 `expr` 函数中判断出*为指针还是乘号， -为负号还是减号。 之后调用 `caculation` 函数进行计算， `caculation` 函数递归求值的主题框架 指导手册里已经给出， 只要按照需求实现判断括号的 `checkParentheses()`函数 和 寻找主运算符的 `findOperator()`函数即可。

```
bool checkParentheses(int l, int r){
    if(tokens[l].type == TK_LB && tokens[r].type == TK_RB){
        int n = 0; //记录除初始括号外单数的左括号数目，每次配对减一，小于0的时候返回false，以此来保证括号起始位置和结束位置配对
        for(int i = l + 1; i < r; i++){
            if(tokens[i].type == TK_LB) n++;
            else if(tokens[i].type == TK_RB) n--;
            if(n < 0){
                return false;
            }
        }
        if(n == 0)
            return true;
    }
    return false;
}
```

```
int findOperator(int left, int right){
    int min_level = 8;
    int op = left;
    int n = 0;
    for(int i = left; i <= right; i++){
        if(tokens[i].type == TK_NUMBER || tokens[i].type == TK_HEX || tokens[i].type == TK_REGISTER)
            continue;
        else if(tokens[i].type == TK_LB)
            n++;
        else if(tokens[i].type == TK_RB)
            n--;
        if(n != 0) //括号未结束
            continue;
        if(tokens[i].level < min_level){
            min_level = tokens[i].level;
            op = i;
        }
    }
    return op;
}
```

整个计算过程需要注意的是，在判断*和-时，注意判断寄存器和括号，另外在正则匹配>和>=号时，需要优先匹配>=号，否则会被忽略掉。

6) 监视点设置和删除

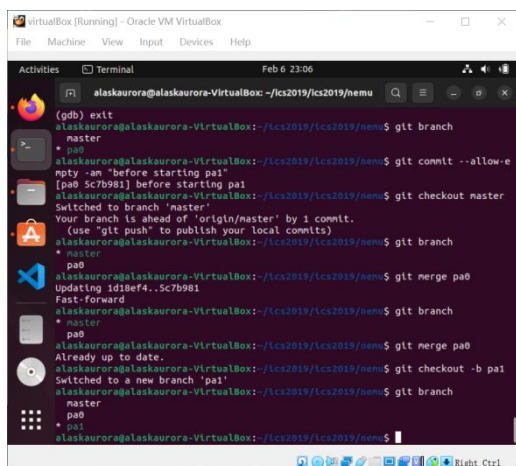
监视点设置和删除主要实现的即链表的增删查功能，比较简单。增加部分如下：

```
WP* res = free_;
free_ = free_ -> next;
res -> next = head;
head = res;
res->value = v;
strcpy(res->expression, args);
return res;
```

删除部分如下：

```
WP *wp = &wp_pool[n];

bool success = false;
WP *p = head;
if(p->NO == wp->NO){
    head = head->next;
    wp->next = free_;
    free_ = wp;
    success = true;
}
else{
    while(p != NULL){
        if(p->next->NO == wp->NO){
            p->next = p->next->next;
            wp->next = free_;
            free_ = wp;
            success = true;
            break;
        }
    }
}
```



创建 git 分支，将各个 pa 分离开来。


```
./build/riscv32-nemu -l ./build/nemu-log.txt -d /media/sf_PA/ics2019/nemu/tools/
qemu-diff/build/riscv32-qemu-so
[src/monitor/monitor.c:36,load_img] No image is given. Use the default build-in
image.
[src/memory/memory.c:16,register_pmem] Add 'pmem' at [0x80000000, 0x87ffffff]
[src/monitor/monitor.c:20,welcome] Debug: ON
[src/monitor/monitor.c:21,welcome] If debug mode is on, A log file will be gener
ated to record every instruction NEMU executes. This may lead to a large log fil
e. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c:28,welcome] Build time: 15:48:39, Jan 14 2021
Welcome to riscv32-NEMU!
For help, type "help"
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single Step Execute
info - Print details of register || watchpoint
x - Scan memory
p - Expression Evaluation
w - Set a New Watchpoint
d - Delete Watchpoint
(nemu)
```

进入简易调试器后，输入 help 显示各个指令对应的信息

```
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
si - Single Step Execute
info - Print details of register || watchpoint
x - Scan memory
p - Expression Evaluation
w - Set a New Watchpoint
d - Delete Watchpoint
(nemu) w $t0
Set Watchpoint Succeed
(nemu) w $t1
Set Watchpoint Succeed
(nemu) w $t2
Set Watchpoint Succeed
(nemu) w $t3
Set Watchpoint Succeed
(nemu) info w
NO      EXPR      VALUE
3       $t3       0
2       $t2       0
1       $t1       0
0       $t0       0
(nemu)
```

设置四个监视点后，使用 info 打印监视点信息，

```
(nemu) d 3
Delete No.3 Watchpoint-
(nemu) info w
NO      EXPR      VALUE
2       $t2       0
1       $t1       0
0       $t0       0
(nemu)
```

删除 3 号监视点后打印监视点信息。

```
(nemu) d 3
Delete No.3 Watchpoint-
(nemu) info w
NO      EXPR      VALUE
2       $t2       0
1       $t1       0
0       $t0       0
(nemu) si 2
00100000: b7 02 00 00          lui 0x80000,t0
watchpoint:Status Changed
(nemu) info w
NO      EXPR      VALUE
2       $t2       0
1       $t1       0
0       $t0      -2147483648
(nemu) si 2
00100004: 23 a0 02 00          sw 0(t0),50
00100008: 03 a5 02 00          lw 0(t0),a0
(nemu) si 2
0010000c: 6b 00 00 00          nemu trap
nemu: HIT GOOD TRAP at pc = 0x0010000c
[src/monitor/cpu-exec.c:32,monitor_statistic] total guest instructions = 4
```

使用命令 si 执行两步，由于寄存器 t0 的值发生改变，程序暂停，使用 info 命令显示监视点的值，发现 t0 寄存器的值变为一个负数，实际上是十六进制 0x80000000，继续使用 si 单步执行，程序最终会 HIT GOOD TRAP

```
(nemu) p (1+2)*(4/3)
3
(nemu) p --1
1
(nemu) p (3/3)+(123*4
wrong expression
```

使用 p 命令进行表达式求值，对于正确的表达式会求出其值，对于错误的表达式会给出相应的提示

3.3PA2:

- 1) 首先实现未实现的 rtl：
取反

```
static inline void rtl_not(rtlreg_t *dest, const rtlreg_t* src1) { //取反
    // dest <- ~src1
    //TODO();
    *dest = ~(*src1);
}
```

变量赋值

```
static inline void rtl_sext(rtlreg_t* dest, const rtlreg_t* src1, int width) { // 变量赋值
    // dest <- signext(src1[(width * 8 - 1) .. 0])
    //TODO();
    switch (width)
    {
    case 4:
        *dest = *src1;
        break;
    case 2:
        *dest = (int16_t)*src1;
        break;
    case 1:
        *dest = (int8_t)*src1;
        break;
    default:
        break;
    }
}
```


入栈

```
static inline void rtl_push(const rtlreg_t* src1) { //源操作数入栈
    // esp <- esp - 4
    // M[esp] <- src1
    cpu.esp -= 4;
    //printf("push 0x%x\n", *src1);
    vaddr_write((cpu.esp), (*src1), 4);
    //TODO();
}
```

出栈

```
static inline void rtl_pop(rtlreg_t* dest) { //出栈
    // dest <- M[esp]
    // esp <- esp + 4
    *dest = vaddr_read(cpu.esp, 4);
    //printf("pop 0x%x\n", *dest);
    cpu.esp += 4;
    //TODO();
}
```

更新标志位

```
static inline void rtl_update_ZF(const rtlreg_t* result, int width) {
    // eflags.ZF <- is_zero(result[width * 8 - 1 .. 0])
    //TODO();
    static uint32_t flags_zf_masks[] = {0, 0xff, 0xffff, 0, 0xffffffff};
    //printf("r:0x%x f:0x%x r&f:0x%x\n", (*result), flags_zf_masks[width], (*result) & flags_zf_masks[width]);
    cpu.EFLAGS.ZF = !((*result) & flags_zf_masks[width]);
}

static inline void rtl_update_SF(const rtlreg_t* result, int width) {
    // eflags.SF <- is_sign(result[width * 8 - 1 .. 0])
    //TODO();
    static uint32_t flags_sf_masks[] = {0, 0x80, 0x8000, 0, 0x80000000};
    cpu.EFLAGS.SF = ((*result) & flags_sf_masks[width]) != 0;
}

static inline void rtl_update_ZFSF(const rtlreg_t* result, int width) {
    rtl_update_ZF(result, width);
    rtl_update_SF(result, width);
}
```

2) 参照 i386 手册，在 opcode table 中填写需要用到的指令并在 all-instr.h 头中添加需要实现的执行函数，分别在 exec 目录下的各个文件中实现这些执行函数。

3) 完成 klib 中的 printf 函数和 string.c 中的函数

4) 在上述过程中，极易出现译码填写或执行函数编写错误的情况，由于一开始没有完成 difftest 模块，每个 bug 调试起来都需要几个小时的时间，所以完成 difftest 模块，在 common.h 中定义 DIFFTEST，在 diff-test.c 中实现 difftest，代码如下：

```

// TODO: Check the registers state with the reference design.
// Set 'nemu_state' to 'NEMU_ABORT' if they are not the same.
//printf("test");
Assert(ref_r.eax == cpu.eax, "eax should be 0x%x, not 0x%x", ref_r.eax, cpu.eax);
Assert(ref_r.ebp == cpu.ebp, "ebp should be 0x%x, not 0x%x", ref_r.ebp, cpu.ebp);
Assert(ref_r.ebx == cpu.ebx, "ebx should be 0x%x, not 0x%x", ref_r.ebx, cpu.ebx);
Assert(ref_r.ecx == cpu.ecx, "ecx should be 0x%x, not 0x%x", ref_r.ecx, cpu.ecx);
Assert(ref_r.edi == cpu.edi, "edi should be 0x%x, not 0x%x", ref_r.edi, cpu.edi);
Assert(ref_r.edx == cpu.edx, "edx should be 0x%x, not 0x%x", ref_r.edx, cpu.edx);
Assert(ref_r.eip == cpu.eip, "eip should be 0x%x, not 0x%x", ref_r.eip, cpu.eip);
Assert(ref_r.esi == cpu.esi, "esi should be 0x%x, not 0x%x", ref_r.esi, cpu.esi);
Assert(ref_r.esp == cpu.esp, "esp should be 0x%x, not 0x%x", ref_r.esp, cpu.esp);
/*Assert(ref_r.EFLAGS.CF == cpu.EFLAGS.CF, "CF should be 0x%x, not 0x%x", ref_r.EFLAGS.CF, cpu.EFLAGS.CF);
Assert(ref_r.EFLAGS.OF == cpu.EFLAGS.OF, "OF should be 0x%x, not 0x%x", ref_r.EFLAGS.OF, cpu.EFLAGS.OF);
Assert(ref_r.EFLAGS.SF == cpu.EFLAGS.SF, "SF should be 0x%x, not 0x%x", ref_r.EFLAGS.SF, cpu.EFLAGS.SF);
Assert(ref_r.EFLAGS.ZF == cpu.EFLAGS.ZF, "ZF should be 0x%x, not 0x%x", ref_r.EFLAGS.ZF, cpu.EFLAGS.ZF);
*/

```

5) 完成 src/device 目录下的 input.c,timer.c,video.c 中的功能函数， 将设备抽象为 IOE

```

size_t video_read(uintptr_t reg, void *buf, size_t size) {
    switch (reg) {
        case _DEVREG_VIDEO_INFO: {
            _VideoInfoReg *info = (_VideoInfoReg *)buf;
            uint32_t screen;
            screen = inl(SCREEN_PORT); //0x26214700
            //printf("screen: 0x%x\n", screen);
            info->width = screen >> 16;
            info->height = screen & 0xffff;
            return sizeof(_VideoInfoReg);
        }
    }
    return 0;
}

```

完成以上功能之后， 即可调试运行 microbench 以及打字游戏等测试程序

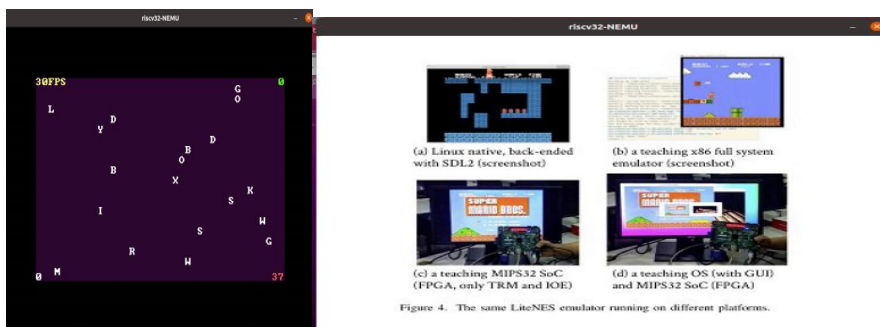
到此为止，PA2 的主体功能实现完毕。

```
[ dlv] PASS!
[ dummy] PASS!
[ fact] PASS!
[ fib] PASS!
[ goldbach] PASS!
[ hello-str] PASS!
[ if-else] PASS!
[ leap-year] PASS!
[ load-store] PASS!
[ matrix-mul] PASS!
[ max] PASS!
[ min3] PASS!
[ mov-c] PASS!
[ movsx] PASS!
[ mul-longlong] PASS!
[ pascal] PASS!
[ prime] PASS!
[ quick-sort] PASS!
[ recursion] PASS!
[ select-sort] PASS!
[ shift] PASS!
[ shutxianhua] PASS!
[ string] PASS!
[ sub-longlong] PASS!
[ sum] PASS!
[ switch] PASS!
[ to-lower-case] PASS!
[ unalign] PASS!
[ wanshu] PASS!
```

microbench 测试的运行结果

```
min time: 515 ms [877]
md5] MD5 digest: * Passed.
min time: 6545 ms [263]
=====
microBench PASS      609 Marks
                    vs. 100000 Marks (i7-7700K @ 4.20GHz)
total time: 35846 ms
emu: HIT GOOD TRAP at pc = 0x801041e0
```

游戏界面



3.4PA3

要实现自陷操作，要完成自陷操作，首先要实现自陷指令，需要实现的指令有 csrrs、csrrw、ecall 和 sret，通过 ecall 指令进入自陷操作，csrrs 和 csrrw 指令对控制状态寄存器进行修改，sret 指令用于自陷后返回，然后需要在 intr.c 文件中进行 raise_intr 函数的编写，raise_intr 函数用于模拟相应过程：将当前 PC 值保存到 sepc 寄存器，在 scause 寄存器中设置异常号，从 stvec 寄存器中取出异常入口地址，跳转到异常入口地址；触发自陷操作后，需要保存上下文，根据 trap.S 汇编代码的压栈的顺序重构 Context 成员体结构，接着要实现正确的事件分发，需要在 __am_irq_handle 函数中通过异常号识别出自陷异常，在 do_event 函数中识别出自陷事件 EVENT_YIELD，最后通过 sret 指令返回并恢复上下文。PA3.2 需要实现用户程序的加载和系统调用，支撑 TRM 程序的运行，为了加载

用户程序首先要实现 loader 函数，因为目前还没有实现文件系统，所以直接在 loader 函数中通过 ramdisk_read 函数把可执行文件中的代码和数据放置在正确的内存位置，然后跳转到程序入口，接着需要完成系统调用的实现，和 3.1 中识别出自陷事件类似，让 nemu 能够识别出系统调用，然后在 do_event 中添加 do_syscall 的调用，根据 nanos.c 中的 ARGS_ARRAY 在 riscv32-nemu 中实现正确的 GPR 宏，随后添加 SYS_yield、SYS_read、SYS_write 和 SYS_brk 系统调用，完成函数_sbrk 实现堆区管理。

PA3.3 需要实现文件系统，添加设备的支持，最终能够运行仙剑奇侠传。首先需要实现函数 fs_open, fs_read 和 fs_close，因为 ramdisk 中的文件数量增加之后，就不适合直接在 loader 函数中直接使用 ramdisk_read，在完成了这几个 fs 函数后，替换掉 loader 函数中的 ramdisk_read 函数，修改其逻辑这样就可以在 loader 中使用文件名来指定加载的程序了，随后完成 fs_write 和 fs_lseek 函数，使其能够进行输出，完成这些函数后，要补充相关的系统调用；要实现虚拟文件系统 VFS，把 IOE 抽象成文件，首先需要在 VFS 中补充多种特殊文件的支持，接着实现函数 serial_write,完成串口的写入，然后完成实现 events_read 函数，支持读操作，最后需要完成 init_fs、fb_write、fb_sync_write、init_device、dispinfo_read 函数，实现对 VGA 设备的支持。要注意的是因为在 Finfo 结构中添加了读函数指针和写函数指针，所以要修改修改 fs_write 和 fs_read 的逻辑，完成这些以后，如果没有错误，就能够运行仙剑奇侠传了。

重新组织 arch.h 中的 Context 结构体使之与 trap.S 中的上下文对应

```
struct _Context {
    uintptr_t esi, ebx, eax, eip, edx, err, eflags, ecx, cs, esp, edi, ebp;
    struct _Protect *prot;
    int irq;
};
```

```
#----|-----entry-----|-----errorcode-----|---irq id---|---handler---|
.globl vecsys;    vecsys:    pushl $0;    pushl $0x80; jmp asm_trap
.globl vectrap;    vectrap:    pushl $0;    pushl $0x81; jmp asm_trap
.globl irq0;        irq0:    pushl $0;    pushl $32; jmp asm_trap
.globl vecnull;    vecnull:    pushl $0;    pushl $-1; jmp asm_trap
```

在 cte.c 中完成 irq.c 实现对 0x8 系统调用和 0x81 自陷的事件分发

```
if (user_handler) {
    _Event ev;
    switch (tf->irq) {
        case 0x80: ev.event = _EVENT_SYSCALL; break;
        case 0x81: ev.event = _EVENT_YIELD; break;
        default: ev.event = _EVENT_ERROR; break;
    }
}
```

在 irq.c 中完成对自陷的处理


```

static Context* do_event(Event e, Context* c) {
    switch (e.event) {
        case EVENT_SYSCALL: do_syscall(c); break;
        case EVENT_YIELD: printf("Yield %d\n", e.event); break;
        default: panic("Unhandled event ID = %d", e.event);
    }

    return NULL;
}

```

到此完成 PA3.1

实现 loader 函数，将从 0 开始，size 为 ramsize 的内存读取到 0x4000000 位置即可。

```

static uintptr_t loader(PCB *pcb, const char *filename) {
    ramdisk_read((void *)DEFAULT_ENTRY, 0, get_ramdisk_size());
    //Log("done ramdisk read");
    //TODO();
    //int fd = fs_open(filename, 0, 0);
    //printf("fd=%d\n", fd);
    //fs_read(fd, (void*)DEFAULT_ENTRY, fs_size(fd));
    //fs_close(fd);

    return DEFAULT_ENTRY;
}

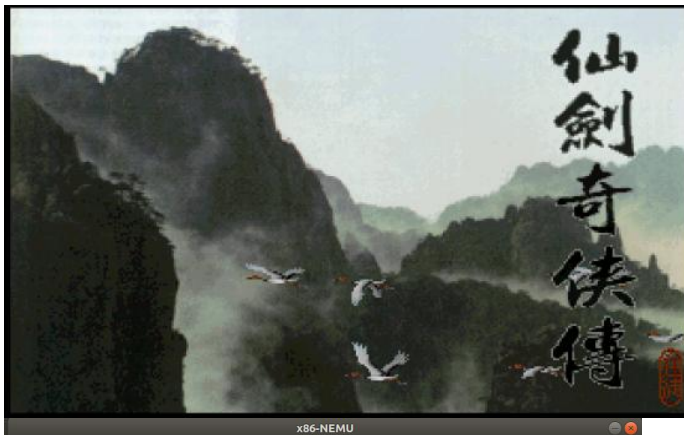
```

```
case SYS_write:
    c->GPRx = fs_write(a[1], (void*)a[2], a[3]);
    break;
```

```
switch(fd){
    case FD_STDIN:
        break;
    case FD_STDOUT:
    case FD_STDERR:
        file_table[fd].write(buf, 0, len);
        break;
```

修改 makefile，从 ramdisk 中读取 helloworld 程序进调试。
PA3.2 到此结束。

实现 fs_open，fs_read，fs_write 等函数，修改 loader 读取文件。
修改 makefile，从 ramdisk 中读取 text 程序进行调试。



3.5 问题解答

XLEN-1	XLEN-2			23	22	21	20	19	18	17
SD	0			TSR	TW	TVM	MXR	SUM	MPRV	
1	XLEN-24			1	1	1	1	1	1	1

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	0	SPP	MPIE	0	SPIE	UPIE	MIE	0	SIE	UIE				
2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1

图 2.6 mstatus 寄存器结构

使用命令 `find . -name "*[.h|.c]" |xargs cat|wc -l` 得出来的行数是 5877。

使用命令 `find . -name "*[.h|.c]" |xargs cat|grep -v ^$|wc -l` 过滤空行后，得出的行数是 4822。

-Wall 的作用是打开 gcc 所有警告。

-Werror 的作用是要求 gcc 将所有警告当成错误处理。

1) 一条指令在 nemu 中执行的过程：

首先根据 PC 值通过 `instr_fetch` 函数取指令，从选出的指令中选取其 opcode，在 `opcode_table` 中进行索引，找到该指令对应的译码辅助函数和执行辅助函数，随后通过译码辅助函数进行译码，将译码得到的相关信息保存在 `decinfo` 中，接着通过执行辅助函数执行指令，执行辅助函数通过 `rtl` 指令对译码得到的信息进行相关操作，计算、读取、保存等等，最后通过 `update_pc` 函数更新 PC 值。

2) 以 `rtl_li` 函数为例单独去掉 `static` 和单独去掉 `inline` 进行重新编译，都不会报错，但将两者同时去掉时，就会报错。原因是都去掉时，在另一个文件中也有对 `rtl_li` 的定义，会出现重复定义的错误，而具有 `static` 关键字时，函数会被限制在本文件内，不会出现重复定义的错误，具有 `inline` 关键字时，函数在预编译时就会展开，不会出现重复定义的错误，但如果将两者同时去掉，就会出现重复定义的错误。

3) 添加后，使用 `grep` 命令查看，共有 81 个 `dummy` 实体；继续添加后，重新编译运行，使用 `grep` 命令，共有 82 个 `dummy` 实体；修改代码后，重新编译报错，原因是两个都初始化后，会产生两个强符号，导致错误。

4) 敲入 `make` 后，会将 `makefile` 文件中第一个目标文件作为最终的目标文件，如果文件不存在，或是文件所依赖的后面的.o 文件的修改时间比这个文件晚，就会重新编译；如果目标文件依赖的.o 文件也不存在，就根据这个.o 文件的生成规则生成，然后生成上一层.o 文件，中间某一步出错就会直接报错。

1) `c` 指向的上下文结构 `_Context` 是在执行自陷操作时，通过 `trap.S` 的汇编程序运行进行赋值的。`riscv-nemu.h` 定义了相关的结构，`trap.S` 对上下文结构体进行赋值，讲义讲清了流程，实现的指令使自陷操作顺利执行。

2) `Nanos-lite` 调用中断，操纵 `AM` 发起自陷指令的汇编代码，随后保存上下文，转入 CPU 自陷指令的内存区域，执行完毕后，恢复上下文，返回运行

时环境。

3) `hello.c` 被编译成 ELF 文件后，位于 `ramdisk` 中，通过 `naive_upload` 函数读入内存并放在正确的位置，交给操作系统进行调用执行，它通过 `SYS_write` 系统调用来输出字符，程序执行完毕后操作系统会回收其内存空间。

4) 操作系统通过库函数读出画面的像素信息，画面通过 `VGA` 输出，`VGA` 被抽象成设备文件，`fs_wrtie` 函数在一步步执行中调用了 `draw_rect` 函数，`draw_rect` 函数把像素信息写入到 `VGA` 对应的地址空间中，最后通过 `update_screen` 函数将画面显示在屏幕上。

4.设计心得与体会

本次课设我在做的过程中,遇到了很多困难的文体。不过克服困难,我觉得是一件非常有意义的事情。

PA1 的整个核心思想就是实现一个编译器,最难的点可能就是实现四则运算,遇到了各种各样的玄学 bug。

PA2 就变得十分类似于组原实验了,对于 `riscv32` 的指令的不熟悉让我非常尴尬,也翻阅了大量资料去解决每一个 bug。

PA3 开始就变得十分精彩了,因为有游戏可以玩了,实现了自陷指令、系统调用、文件系统

这次 PA 串联了我们在大学前三年里学习的很多内容,从 PA1 中基本的程序设计和算法,到 PA2 中涉及到的汇编语言、组成原理等,再到 PA3 和 PA4 中的操作系统。通过完成 PA 的内容,我回顾了很多以前学习过的知识,对其中的一些内容也有了新的认识,但我认为这门课程还是存在很多需要改进的地方。

,我希望这门课程设计可以增加一些自动测试和评分的系统,就像组成原理实验与课设中使用的 `educoder` 自动测试那样。对于 PA1 的内容,主要是程序设计和算法的内容,使用大量的样例进行测试即可;对于 PA2 的内容,由于指令执行的结果具有确定性,因此比较每条指令执行后的处理器状态就可以验证对错。这两部分,虽然我们在实验中也实现了 `qemu-diff` 这样的功能,但是倘若老师能够提供一套评测系统,我们在实验中可以更加准确的对实现结果正确与否进行认定。至于 PA3 和 PA4 这些与操作系统相关的内容,参考课程 6.828 的设计,同样可以提供一套测试评分功能。我认为,对于 PA 这种以功能复现为目的的实验,自动测评对我们的实验可以起到很大的帮助,它能够让我们准确的判断和掌握实验进度,不至于到了实验后期才发现前面的 bug,导致大量的更改。

