华中科技大学

2024

系统能力培养　课程实验报告

题　　目：	指令模拟器
专　　业：	计算机科学与技术
班　　级：	CS2103 班
学　　号：	U202115395
姓　　名：	杜启铭
电　　话：	13526877089
邮　　件：	2454235539@qq.com
完成日期：	2024-10-24

计算机科学与技术学院

目 录

1　课程实验概述

1.1 课设目的

本次课程设计通过实现一个经过简化但功能完备的 riscv32 模拟器 NEMU，最终在 NEMU 上运行游戏“仙剑奇侠传”，来探究“程序在计算机上运行”的基本原理。

1.2 课设任务

本次课程设计主要包含下列实验内容。

1. 实现简易调试器、表达式求值、监视点与断点等功能。

2. 运行一个 C 程序、丰富指令集并测试所有程序、实现 I/O 指令并测试打字游戏。

3. 实现系统调用、实现文件系统、运行仙剑奇侠传。

4. 实现分页机制、实现进程上下文切换、时钟中断驱动的上下文切换。

1.3 实验环境

使用教师提供的 virtual box 镜像。

2 PA1-开天辟地的篇章: 最简单的计算机

2.1 简单调试器

我们需要为 nemu 实现一些简单的调试功能，包括单步执行、打印程序状态、扫描内存、表达式求值、扫描内存、设置监视点、删除监视点。

2.1.1 单步执行 si

cpu_exec 函数实现了执行指定次数的 CPU 循环，因此单步执行的功能通过直接调用该函数即可实现。

2.1.2 打印程序状态 info

程序状态包括两种状态：寄存器状态和监视点状态。

定义一个名为 cmd_info 的函数，用于处理 info 命令。根据传入的参数 args，如果参数是 r，则调用 isa_reg_display() 显示寄存器信息；如果参数是 w，则调用 display_watchpoints() 显示监视点信息并返回；如果参数无效，则打印 "Wrong argument!" 错误信息。如果没有提供参数，则打印 "Lack argument!" 错误信息。

2.1.3 扫描内存 x

定义一个名为 cmd_x 的函数，用于处理 x 命令。该函数从参数 args 中解析出要读取的内存单元数量 number 和起始地址 index，如果解析失败或参数无效，则打印错误信息 "Wrong argument!" 并返回。否则，它会循环读取指定数量的内存单元，每读取四个单元打印一行，

调用 isa_vaddr_read 函数读取内存地址的值，并以十六进制格式打印地址和值。

2.2 表达式求值

2.2.1 表达式解析

定义一个名为 make_token 的函数，用于将输入字符串 e 分解成一系列的 tokens。函数通过正则表达式逐个匹配输入字符串中的子串，并根据匹配的规则将子串记录为相应类型的标记。对于不同类型的标记，函数会执行不同的操作。

2.2.2 表达式求值

定义一个名为 calculate 的递归函数，用于计算表达式的值。函数根据传入的标记数组 tokens 和索引范围 [i, j] 解析并计算表达式的值。如果 i 和 j 相等且标记是数字或寄存器，则返回其值；如果表达式被括号包围，则去掉括号递归计算；否则，找到主操作符并递归计算其左右操作数的值，并根据操作符类型执行相应的运算。函数通过 success 标志指示计算是否成功，并处理各种运算符和错误情况，如除零错误和无效表达式。

2.3 设置与删除监视点

我们需要实现：监视点的创建、删除、打印、检查。

监视点的创建。定义一个名为 new_wp 的函数，用于分配一个新的监视点（WP）。如果没有可用的空闲监视点，则打印错误信息并触发断言失败。否则，从空闲链表中取出一个监视点，将其插入到活动

监视点链表的头部，并返回该监视点的指针。

监视点的删除。定义一个名为 free_wp 的函数，用于根据监视点编号 NO 从活动监视点链表中删除相应的监视点；如果找到该监视点，则将其从链表中移除并添加到空闲监视点链表中，以便后续复用

监视点的打印。定义一个名为 display_watchpoints 的函数，用于显示当前所有活动的监视点。函数首先打印表头，然后遍历活动监视点链表 head，对于每个监视点，打印其编号 (NO)、表达式 (wp_expr) 和上次计算的结果 (last_value)。

监视点的检查。定义一个名为 check_watchpoints 的函数，用于检查所有活动的监视点是否发生变化。函数遍历监视点链表 head，计算每个监视点表达式的当前值 res，如果计算失败则打印错误信息并触发断言失败；如果当前值与上次记录的值 last_value 不同，则打印监视点信息和新旧值，并更新 last_value。函数返回一个布尔值 result，指示是否有监视点发生了变化。

2.4 运行结果

简单调试器的功能已基本实现。

2.5 必答题

这里我们来对实验文档中的必答题进行逐一的解答。

1. 我选择的 ISA 是__riscv32__。

2. 用于调试的时间为 15 小时，实现简单调试器后的调试时间将降低为 3 小时，由此可见，通过实现一定的基础设施，可以有效的减

少我们在后续工作中 debug 的工作量。

3. riscv32 的指令格式有：R 型（Register）、I 型（Immediate）、S 型（Store）、B 型（Branch）、U 型（Upper Immediate）、J 型（Jump）

LUI（Load Upper Immediate）指令将一个 20 位的立即数加载到寄存器的高 20 位，低 12 位填 0。

mstatus 包含字段：MIE 机器模式全局中断使能、MPI 机器模式中断使能前值、MPP 机器模式前模式、FS 浮点状态、XS 扩展状态、SD 状态脏位。

4.

（1）nemu/目录下的所有.c 和.h 文件总共有多少行代码?

使用以下命令统计代码行数：

```
find nemu/ -name *.c -o -name *.h | xargs wc -l
```

（2）和框架代码相比, 你在 PA1 中编写了多少行代码?

两次使用（1）中的命令并计算结果。

（3）将统计代码行数的命令写入 Makefile 中。

在 Makefile 添加以下内容：

```
count:
    find nemu/ -name *.c -o -name *.h | xargs wc -l
```

运行 make count 即可统计代码行数。

（4）除去空行之外, nemu/目录下的所有.c 和.h 文件总共有多少行代码?

使用以下命令统计除去空行的代码行数：

```
find nemu/ -name *.c -o -name *.h | xargs grep -v ^\s*$$ | wc -l
```

5. -Wall 和-Werror 是 GCC 编译器中的两个常用选项：

-Wall：

（1）启用所有常见的警告选项，帮助开发者发现潜在的代码问题。

（2）包括未使用的变量、未初始化的变量、隐式函数声明等警告。

-Werror：

（1）将所有警告视为错误，编译器在遇到警告时会停止编译。

（2）强制开发者修复所有警告，确保代码质量。

为什么要使用-Wall 和-Werror：

（1）提高代码质量：通过启用警告，开发者可以发现并修复潜在的代码问题。

（2）强制修复警告：将警告视为错误，确保所有警告都被修复，避免潜在的运行时错误。

（3）保持代码整洁：减少代码中的潜在问题和不良实践，保持代码库的整洁和可维护性。

3　PA2-简单复杂的机器：冯诺依曼计算机系统设计

3.1 在 NEMU 中运行第一个 C 程序 dummy

首先浏览 dummy-riscv32-nemu.txt，找到需要实现的指令有：li、auipc、addi、jal、mv、sw、jalr。

为了实现一条新指令，需要：

（1）在 opcode_table 中填写正确的译码辅助函数，执行辅助函数、操作数宽度。

（2）用 RTL 实现正确的译码辅助函数和执行辅助函数。使用 RTL 伪指令时要遵守小型调用约定。

下面以 LUI 指令为例，介绍译码辅助函数和执行辅助函数的实现及 opcode_table 的填写。LUI（Load Upper Immediate）指令用于将一个立即数加载到寄存器的高 20 位。

opcode_table 是一个包含 32 个 OpcodeEntry 的数组，用于根据指令的 opcode 字段选择相应的译码辅助和执行辅助函数。每个 OpcodeEntry 包含一个译码辅助函数和一个执行辅助函数。

LUI 指令的 opcode 为 0b0110111，对应 opcode_table 中的第 13 个条目。

```
1. static OpcodeEntry opcode_table [32] = {
2.   // 其他条目省略...
3.   /* b01 */ IDEX(st, store), EMPTY, EMPTY, EMPTY, IDEX(R, r), IDE
   X(U, lui), EMPTY, EMPTY,
4.   // 其他条目省略...
5. };
```

U 是译码辅助函数，用于解析 U 型指令。

lui 是执行辅助函数，用于执行 LUI 指令。

译码辅助函数用于解析指令并提取操作数。对于 LUI 指令，我们需要解析立即数和目标寄存器。

```
make_DHelper(U) {
  // 解析立即数并左移 12 位
  decode_op_i(id_src, decinfo.isa.instr.imm31_12 << 12, true);
  // 解析目标寄存器
  decode_op_r(id_dest, decinfo.isa.instr.rd, false);
  // 打印操作数信息
  print_Dop(id_src->str, OP_STR_SIZE, "0x%x", decinfo.isa.instr.imm31_12);
}
```

执行辅助函数用于执行指令的具体操作。对于 LUI 指令，我们需要将立即数加载到目标寄存器的高 20 位。

```
static inline void exec_lui(DecodeExecState *s) {
  // 将立即数加载到目标寄存器
  rtl_li(&s->dest->val, s->src1->imm);
}
```

3.2 实现指令，运行所有 cputest

1. 剩余指令的实现

与 LUI 指令的实现类似，剩余指令的实现大致相同。指令的实现很繁琐，需要的时间很多。

2. 字符串处理函数的实现。

待实现的都是比较常见的函数，不再赘述。

3. printf 的实现。

printf 函数

```
int printf(const char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  char buf[1024] = {0};
  int cnt = vsprintf(buf, fmt, ap);
```

```
6.    for(int i = 0; i < cnt; i++) {
7.      _putc(buf[i]);
8.    }
9.    va_end(ap);
10.   return cnt;
11.}
```

（1）使用 va_list 类型定义 ap 来存储可变参数。

（2）调用 va_star 初始化 ap。

（3）定义一个缓冲区 buf 来存储格式化后的字符串。

（4）调用 vsprintf 将格式化字符串和参数写入 buf。

（5）使用_putc 将 buf 中的字符逐个输出。

（6）调用 va_end 结束可变参数处理。

（7）返回写入的字符数。

vsprintf 函数

```
1. int vsprintf(char *out, const char *fmt, va_list ap) {
2.   int cnt = 0;
3.   for(int i = 0; fmt[i]; i++) {
4.     if(fmt[i] != '%') {
5.       out[cnt++] = fmt[i];
6.     } else {
7.       i++;
8.       switch(fmt[i]) {
9.         case 'd': ...
10.        case 's': ...
11.        case 'c': ...
12.        case 'x': ...
13.        case 'u': ...
14.        default: break;
15.      }
16.    }
17.  }
18.  out[cnt] = '\0';
19.  return cnt;
20.}
```

（1）初始化字符计数器 cnt

（2）遍历格式化字符串 fmt。

（3）如果当前字符不是%，直接将其写入 out。如果是%且下一个字符是 d/s/c/x/u，相应地处理后续的格式化字符。

sprintf 函数和 snprintf 函数的实现与 printf 函数的实现大致相同。

3.3 输入输出

（1）实现_DEVREG_TIMER_UPTIME 的功能

实现两个主要函数：__am_timer_read 和 __am_timer_init。

__am_timer_read 根据传入的寄存器地址读取定时器信息，支持读取系统启动时间_DEVREG_TIMER_UPTIME 和当前日期 _DEVREG_TIMER_DATE。

__am_timer_init 初始化定时器，记录系统启动时的时间。通过访问硬件寄存器 RTC_ADDR，这些函数获取并处理时间信息。

（2）实现_DEVREG_INPUT_KBD 的功能

实一个函数 __am_input_read，用于读取键盘输入。当寄存器地址为 _DEVREG_INPUT_KBD 时，它从键盘地址 KBD_ADDR 读取键盘状态，并将按键状态和按键码存储到 _DEV_INPUT_KBD_t 结构中。

（3）实现_DEVREG_VIDEO_INFO 的功能

实现三个主要函数：

__am_video_read：读取视频信息。根据寄存器地址读取屏幕宽度和高度，并存储在 info 结构中。

__am_video_write：将帧缓冲区中的像素数据写入视频内存。根据寄存器地址判断操作类型。将像素数据逐行拷贝到视频内存。如果需要同步显示，则调用同步函数。

__am_vga_init：初始化 VGA 显示。计算屏幕的总像素数。将帧缓冲区中的每个像素设置为其索引值。调用同步函数以更新显示。

3.4 运行结果

riscv32 大部分指令的功能基本实现。三个输入输出接口成功实现。

3.5 必答题

1. nemu 中一条指令的执行流程可以分为四个步骤：取指、译码、执行、更新 PC。

2. 去掉 static 后，函数的链接属性变为外部链接。这意味着该函数在多个源文件中可能会有多个定义，从而导致链接器错误（重复定义）。

去掉 inline 后，函数不再是内联函数，编译器可能会为每个调用生成一个函数调用，从而增加函数调用的开销。但不会导致编译或链接错误。

3.

（1）在 nemu/include/common.h 中添加 volatile static int dummy; 并重新编译。

结果，由于 dummy 变量被定义为 static，它在每个包含 common.h 的源文件中都是一个独立的实体。因此，编译后的 NEMU 将包含多个 dummy 变量的实体，每个包含 common.h 的源文件都会有一个 dummy 变量。

可以通过以下步骤验证：

1）在 common.h 中添加 volatile static int dummy;。

2）运行 make clean 清理之前的编译结果。

3）运行 make 重新编译 NEMU。

4）使用 nm 命令查看生成的目标文件，检查 dummy 变量的定义。

（2）在 nemu/include/debug.h 中添加 volatile static int dummy; 并重新编译。

结果，由于 dummy 变量被定义为 static，它在每个包含 debug.h 的源文件中也是一个独立的实体。因此，编译后的 NEMU 将包含更多的 dummy 变量的实体，每个包含 common.h 和 debug.h 的源文件都会有一个 dummy 变量。

与上题相比，添加到 debug.h 中的 dummy 变量会增加 dummy 变量的实体数量，因为 debug.h 可能被更多的源文件包含。

（3）为两处 dummy 变量进行初始化

将 common.h 和 debug.h 中的 dummy 变量初始化为 0。然后重新编译 NEMU。

结果，编译器可能会报错，提示重复定义 dummy 变量。这是因为 static 变量在每个包含它的源文件中都是独立的，但初始化会导致链接器错误。

之前没有出现这样的问题是因为 static 变量在每个源文件中都是独立的实体，不会在链接阶段冲突。但一旦进行初始化，编译器会尝试在多个源文件中初始化同一个变量，从而导致链接器错误。

4. 在 nemu 目录下执行 make 后，make 程序会根据 Makefile 的规则组织 .c 和 .h 文件，最终生成可执行文件 nemu/build/$ISA-nemu。以下是一个简要的过程描述：

（1）读取 Makefile

make 程序首先读取 Makefile 文件，并解析其中的变量和规则。

（2）设置目标 ISA

根据 Makefile 中的条件判断，设置目标 ISA（指令集架构）。

```
1. ifneq ($(MAKECMDGOALS),clean) # ignore check for make clean
2. ISA ?= riscv32
3. ISAS = $(shell ls src/isa/)
4. $(info Building $(ISA)-$(NAME))
```

这段代码会设置 ISA 变量为 riscv32，并列出 src/isa/目录下的所有 ISA。

（3）查找源文件

Makefile 会定义源文件的路径和模式，通常会使用通配符查找所有的 .c 文件。

```
1. SRCS = $(wildcard src/**/*.c)
```

这行代码会查找 src/ 目录及其子目录下的所有 .c 文件，并将它们存储在 SRCS 变量中。

（4）编译源文件

Makefile 会定义如何编译每个源文件。通常会使用一个编译规则。

```
1. OBJS = $(SRCS:.c=.o)
2. %.o: %.c
3.     $(CC) $(CFLAGS) -c $< -o $@
```

这段代码会将所有的 .c 文件编译成对应的 .o 文件。

（5）链接目标文件

Makefile 会定义如何将所有的目标文件链接成最终的可执行文件。

```
1. nemu/build/$(ISA)-nemu: $(OBJS)
2.     $(CC) $(LDFLAGS) -o $@ $^
```

这行代码会将所有的 .o 文件链接成最终的可执行文件 nemu/build/$(ISA)-nemu。

（6）生成可执行文件

最终，make 程序会执行所有的编译和链接命令，生成可执行文件 nemu/build/$(ISA)-nemu。

4　PA3-穿越时空的旅程: 批处理系统

4.1 将上下文管理抽象成 CTE

4.1.1 触发自陷操作

1. 文件 nexus-am/am/src/riscv32/nemu/cte.c 中，_yield 函数中执行指令” li a7, -1; ecall”。

2. 文件 nemu/src/isa/riscv32/exec/system.c 中，系统指令的执行辅助函数调用 raise_intr 函数，参数为中断号 NO（自陷的中断号为-1）以及当前的 PC 值。

3. 文件 nemu/src/isa/riscv32/intr.c 中，raise_intr 函数接收两个参数：中断号 NO 和异常程序计数器 epc。函数将 epc 保存到 sepc 寄存器，将 NO 保存到 scause 寄存器，并设置跳转地址为中断向量表 stvec 的地址，最后设置跳转标志以便处理异常或中断。。

4. 文件 nexus-am/am/src/riscv32/nemu/trap.S 中，__am_asm_trap 函数包含如下操作：

（1）调整栈指针，为保存上下文腾出空间。

（2）保存所有通用寄存器和关键的 CSR 寄存器（scause、sstatus、sepc）。

（3）调用中断处理函数 __am_irq_handle。

（4）恢复关键的 CSR 寄存器和所有通用寄存器。

（5）恢复栈指针。

（6）使用 sret 指令返回到异常发生前的状态。

5. 文件 nexus-am/am/src/riscv32/nemu/cte.c 中，定义了一个名为 __am_irq_handle 的函数，用于处理中断和异常。函数接收一个 _Context 结构体指针 c，表示当前的 CPU 上下文。根据 c->cause 的值，函数将事件类型设置为 _EVENT_YIELD、_EVENT_SYSCALL 或 _EVENT_ERROR，然后调用用户定义的 user_handler 函数处理事件。user_handler 在_cte_init 中被初始化为 do_event 函数。

6. 文件 nanos-lite/src/irq.c 中，do_event 函数对传入的时间进行解析，做出相应的操作，对于 yield 操作，我们直接输出一段文本，表示程序运行至此即可。我们现在仅有一个上下文，因此不做上下文切换，此处直接返回 NULL 给__am_irq_handle。

7. 沿着上述调用链逐级返回。

补全上述的函数调用链即可。

4.1.2 保存上下文

重新组织_Context 结构体。根据 trap.S 中的代码很容易可以得到 _Context 结构体中的成员顺序为 gpr、cause、status、epc、as。

4.1.3 事件分发和恢复上下文

先在__am_irq_handle()函数中通过异常号识别出自陷异常，然后将 event 设置为编号为_EVENT_YIELD 的自陷事件。

之后在 do_event()函数中识别出自陷事件_EVENT_YIELD，然后输出“Self trap!”即可。

4.2 用户程序和系统调用

4.2.1 加载第一个用户程序

实现静态函数 loader，用于加载 ELF 格式的可执行文件。

1. 打开文件

使用 fs_open 函数打开指定的文件，并返回文件描述符 fd。

2. 读取 ELF 文件头

使用 memcmp 检查文件头的标识字段是否匹配 ELF 魔数。

3. 遍历程序头表

如果条目的类型是 PT_LOAD，则将文件指针移动到段的偏移位置，读取段内容到内存中的 Phdr.p_vaddr 位置，并使用 memset 将未初始化的数据部分清零。

4. 关闭文件并返回入口地址 Ehdr.e_entry

4.2.2 系统调用

1. 用户程序调用_syscall_函数。

2. _syscall_函数接收四个参数：type、a0、a1 和 a2，分别表示系统调用的类型和三个参数。函数内部使用内联汇编将这些参数加载到特定的寄存器中，然后执行系统调用指令 SYSCALL。系统调用的返回值存储在寄存器 GPRx 中，并返回给调用者。

3. 经过 make_Ehelper 、 raise_intr 、 __am_asm_trap 、 __am_irq_handle、do_syscall 调用链。

补全上述的函数调用链即可。

4.3 文件系统

4.3.1 简易文件系统

1. 实现 fs_open 函数，用于打开文件。

函数接收三个参数：文件路 pathname、标志 flags 和模式 mode。它首先断言 pathname 不为空，然后遍历文件表 file_table，查找与 pathname 匹配的文件名。如果找到匹配的文件名，则返回文件在文件表中的索引。如果找不到匹配的文件，则触发断言失败，表示无法找到文件。

2. 实现 fs_close 函数，用于关闭文件。

函数接收一个文件描述符 fd 作为参数，将文件表 file_table 中对应文件的打开偏移量 open_offset 重置为 0，并返回 0 表示成功关闭文件。

3. 实现 fs_read 函数，用于从文件中读取数据。

函数接收三个参数：文件描述符 fd、缓冲区指针 buf 和读取长度 len。首先，它断言 fd 在有效范围内，然后计算实际读取长度 read_len，确保不超过文件剩余大小。接着，根据文件表中的 read 函数指针选择读取方式（自定义读取函数或 ramdisk_read），并将读取的数据存储到缓冲区 buf 中。最后，更新文件的打开偏移量 open_offset 并返回实际读取的字节数 ret。

4. 实现 fs_write 函数，用于将数据写入文件。

函数接收三个参数：文件描述符 fd、数据缓冲区指针 buf 和写入长度 len。首先，它断言 fd 在有效范围内，然后计算实际写入长

度 write_len，确保不超过文件剩余大小。接着，根据文件表中的 write 函数指针选择写入方式（自定义写入函数或 ramdisk_write），并将数据写入文件。最后，更新文件的打开偏移量 open_offset 并返回实际写入的字节数 ret。

5. 实现 fs_lseek 函数，用于在文件中移动文件指针。

函数接收三个参数：文件描述符 fd、偏移量 offset 和定位方式 whence。根据 whence 的值（SEEK_SET、SEEK_CUR 或 SEEK_END），函数分别将文件指针设置为相对于文件开头、当前位置或文件末尾的偏移量。最后，函数确保文件指针不超过文件大小，并返回新的文件指针位置。

4.3.2 把 IOE 抽象成文件

实现三个函数，用于处理显示信息和帧缓冲区的读写操作：

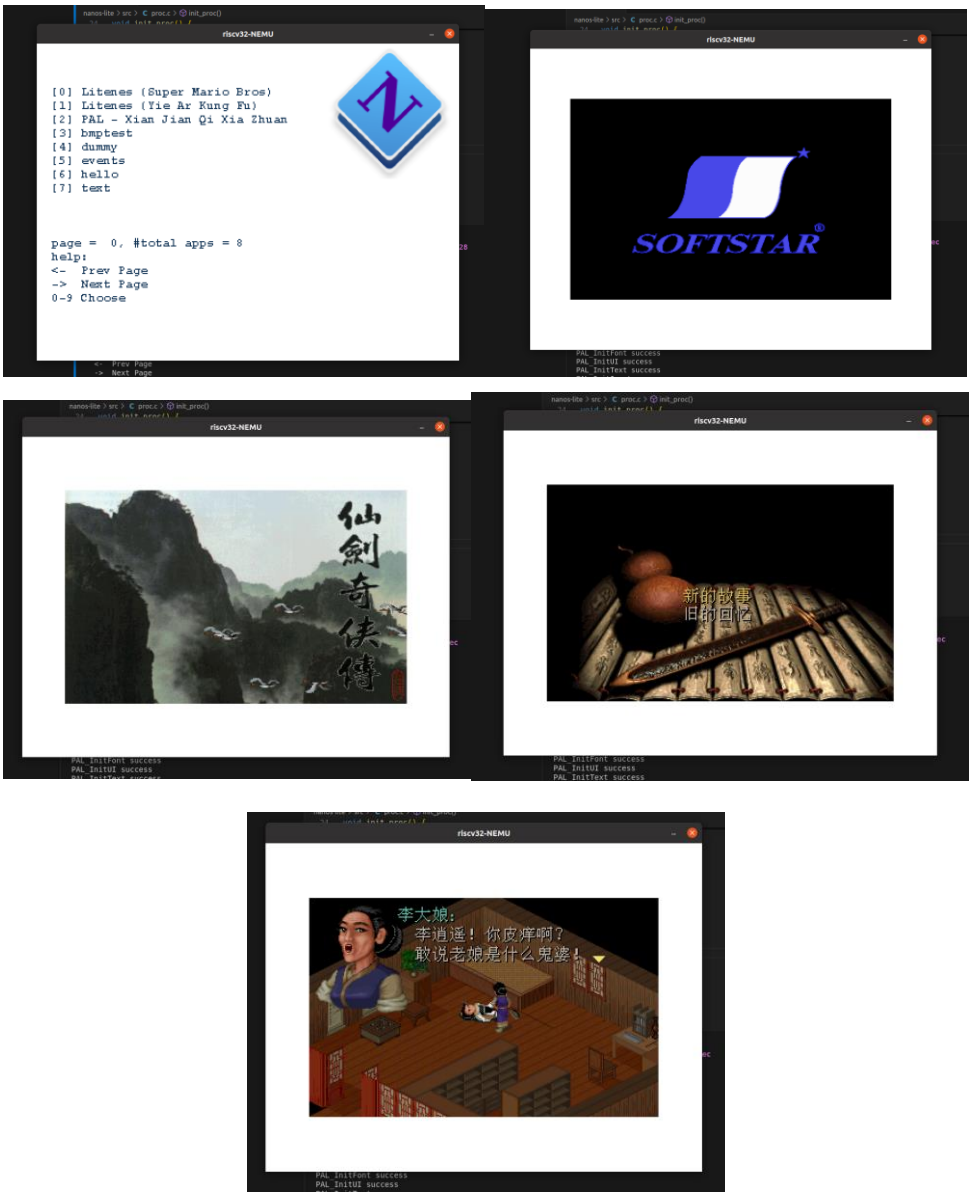
dispinfo_read：将显示信息从指定偏移量开始读取到缓冲区，并返回读取的字节数。

fb_write：将缓冲区中的数据写入帧缓冲区，从指定偏移量开始，计算写入位置的坐标，并调用 draw_rect 函数绘制一行像素。

fbsync_write：调用 draw_sync 函数同步显示。

4.4 运行结果

“仙剑奇侠传”顺利运行。如图所示。



运行结果

4.5 必答题

1. 你会在__am_irq_handle()中看到有一个上下文结构指针 c, c 指向的上下文结构究竟在哪里？这个上下文结构又是怎么来的？具体地，这个上下文结构有很多成员，每一个成员究竟在哪里赋值的？$ISA-nemu.h, trap.S，上述讲义文字，以及你刚刚在 NEMU 中实现的新指令，这四部分内容又有什么联系？

__am_irq_handle() 中的 c 是指向上下文结构的指针。上下文结

构在中断或异常发生时由汇编代码（trap.S）创建和填充。上下文结构的成员在汇编代码（trap.S）中被赋值。riscv32-nemu.h 定义了上下文结构，trap.S 实现了中断处理，NEMU 中的新指令涉及上下文切换。

2. 从 Nanos-lite 调用_yield()开始, 到从_yield()返回的期间, 这一趟旅程具体经历了什么？软(AM, Nanos-lite)硬(NEMU)件是如何相互协助来完成这趟旅程的?

该问题已在 4.1.1 中详细介绍，此处不再赘述。

3. 我们知道 navy-apps/tests/hello/hello.c 只是一个 C 源文件, 它会被编译链接成一个 ELF 文件. 那么, hello 程序一开始在哪里？它是怎么出现内存中的？为什么会出现在目前的内存位置？它的第一条指令在哪里？究竟是怎么执行到它的第一条指令的? hello 程序在不断地打印字符串, 每一个字符又是经历了什么才会最终出现在终端上?

hello 程序通过编译和链接生成一个 ELF 文件，存放于文件系统中。当操作系统加载 hello 程序时，会使用加载器（loader）将 ELF 文件加载到内存中。加载器会解析 ELF 文件的头部信息，确定代码段和数据段在内存中的位置。ELF 文件头部包含一个入口点地址（entry point），这是程序开始执行的第一条指令的地址。加载器会将程序计数器（PC）设置为这个入口点地址。hello 程序中调用 printf 函数来打印字符串。printf 是一个标准库函数，它会将字符串格式化并输出到标准输出。printf 函数内部会调用系统调用将字符串写入标准输出。系统调用会将控制权交给操作系统内核。操作系统内核通过

终端驱动程序将字符发送到终端设备。终端设备可以是一个物理终端或一个虚拟终端。

4. 运行仙剑奇侠传时会播放启动动画, 动画中仙鹤在群山中飞过. 这一动画是通过 navy-apps/apps/pal/src/main.c 中的 PAL_SplashScreen()函数播放的. 阅读这一函数, 可以得知仙鹤的像素信息存放在数据文件 mgo.mkf 中. 请回答以下问题: 库函数, libos, Nanos-lite, AM, NEMU 是如何相互协助, 来帮助仙剑奇侠传的代码从 mgo.mkf 文件中读出仙鹤的像素信息, 并且更新到屏幕上? 换一种 PA 的经典问法: 这个过程究竟经历了些什么?

（1）读取文件：

PAL_SplashScreen() 函数会调用库函数（如 fopen, fread）来打开并读取 mgo.mkf 文件。

库函数会调用 libos 提供的系统调用接口来执行文件操作。

libos 会通过系统调用接口与 Nanos-lite 进行交互，Nanos-lite 会处理文件系统请求，读取文件数据并返回给 libos。

（2）解析像素信息：

PAL_SplashScreen() 函数会解析从 mgo.mkf 文件中读取的像素信息，存储在内存中。

（3）绘制图像：

PAL_SplashScreen() 函数会调用图形库函数来绘制图像。

图形库函数会调用 libos 提供的图形接口，libos 会通过 AM 提供的硬件抽象接口与底层硬件进行交互。

（4）更新屏幕：

AM 会通过帧缓冲区将像素信息写入屏幕。

在模拟环境中，NEMU 会模拟硬件行为，将帧缓冲区中的像素信息显示在模拟器窗口中。

5　PA4-虚实交错的魔法：分时多任务

5.1 上下文切换

1. CTE 的_kcontext()函数

_kcontext 函数，用于创建和初始化一个新的上下文 _Context。它接受一个栈区域 _Area、一个函数指针 entry 和一个参数 arg。函数首先在栈的末尾分配 _Context 结构体，然后将其内存清零，并将 entry 函数的地址设置为上下文的程序计数器 epc。最后返回初始化后的上下文指针。

```
1. _Context *_kcontext(_Area stack, void (*entry)(void *), void *arg
   ) {
2.   _Context *context = stack.end - sizeof(_Context);
3.   memset(context, 0x00, sizeof(_Context));
4.   context->epc = (uint32_t)entry;
5.   return context;
6. }
```

2. Nanos-lite 的 schedule()函数

schedule 函数，用于保存当前上下文指针 prev 到当前进程控制块 current->cp，然后总是选择 pcb[0] 作为新的进程，并返回新的上下文指针 current->cp。

```
1. _Context* schedule(_Context *prev) {
2.   current->cp = prev;
3.   current = &pcb[0];
4.   return current->cp;
5. }
```

3. 在 Nanos-lite 收到_EVENT_YIELD 事件后，调用 schedule()并返回新的上下文。

4. 修改 CTE 中__am_asm_trap()的实现，使得从__am_irq_handle()

返回后, 先将栈顶指针切换到新进程的上下文结构, 然后才恢复上下文, 从而完成上下文切换的本质操作。

只需在从__am_irq_handle 返回后，将 a0 中保存的、将要切换到的目标进程的栈指针赋给 sp 寄存器即可。

5.2 将虚存管理抽象成 VME

1. 完善 isa_vaddr_read、isa_vaddr_write、page_translate 函数

isa_vaddr_read 函数从虚拟地址 addr 读取长度为 len 的数据。如果地址跨页，则逐字节读取并组合成所需长度的数据；否则，通过页表转换为物理地址后直接读取数据。如果分页模式未启用，则直接从物理地址读取数据。

isa_vaddr_write 函数将数据 data 写入虚拟地址 addr。如果地址跨页，则逐字节写入数据；否则，通过页表转换为物理地址后直接写入数据。如果分页模式未启用，则直接将数据写入物理地址。

page_translate 函数将虚拟地址 va 转换为物理地址。它首先通过读取页表项来获取页表地址，并检查 PTE 的有效性。如果 PTE 无效，则打印错误信息并触发断言失败。然后，它通过读取页表中的页地址，并结合虚拟地址的偏移量，计算出最终的物理地址并返回。

2. context_uload、_ucontext 函数的实现

context_uload 函数加载用户程序到指定的进程控制块。它首先将 PCB 清零，然后设置地址空间保护，接着通过 loader 函数加载程序并获取入口地址。随后，它初始化栈区域，并调用 _ucontext 函数创建用户上下文，将入口地址设置为程序的起始地址。

_ucontext 函数创建和初始化用户上下文 _Context。它接受地址空间指针 as、用户栈区域 ustack、内核栈区域 kstack、入口地址 entry 和参数 args。函数首先在用户栈的末尾分配 _Context 结构体，并将其内存清零。然后，它将入口地址 entry 设置为上下文的程序计数器 epc，将地址空间指针 as 赋值给上下文，并将状态寄存器 status 设置为 0x2。最后，返回初始化后的上下文指针。

3. _map 函数的实现

_map 函数将虚拟地址 va 映射到物理地址 pa，并设置相应的保护权限 prot。它首先获取页目录项 pptab，如果该项无效，则分配一个新的页表并标记为有效。然后获取页表项 ptab，如果该项无效，则将物理地址 pa 映射到该页表项并标记为有效。

```
1. int _map(_AddressSpace *as, void *va, void *pa, int prot) {
2.   PTE *pdir = (void *)as->ptr;
3.   PDE *pptab = &pdir[PDX(va)];
4.   if (!(*pptab & PTE_V)) {
5.     *pptab = (PDE *)pgalloc_usr(1);
6.     *pptab = ((*pptab >> 12) << 10) | PTE_V;
7.   }
8.   PDE *ptab = &(((PDE *)PTE_ADDR(*pptab))[PTX(va)]);
9.   if (!(*ptab & PTE_V)) {
10.    *ptab = (((PDE)pa >> 12) << 10) | PTE_V;
11.  }
12.  return 0;
13.}
```

4. 修改__am_irq_handle 函数的实现

在__am_irq_handle 的开头调用__am_get_cur_as，来将当前的地址空间描述符指针保存到上下文中。在__am_irq_handle 返回前调用__am_switch 来切换地址空间，将调度目标进程的地址空间落实到 MMU 中。

5. mm_brk 的实现

mm_brk 函数调整程序的 brk 以分配更多的内存。如果新的断点超出了当前进程的 max_brk，则通过循环分配新的物理页并将其映射到虚拟地址空间中，更新最大断点值。

```
1. int mm_brk(uintptr_t brk, intptr_t increment) {
2.   uintptr_t new_brk = brk + increment;
3.   if (new_brk > current->max_brk) {
4.     uintptr_t begin = PGROUNDUP(current->max_brk);
5.     uintptr_t end = PGROUNDUP(new_brk);
6.     void *va = NULL, *pa = NULL;
7.     for (uintptr_t i = begin; i < end; i += PGSIZE) {
8.       va = (void *)i;
9.       pa = new_page(1);
10.      _map(&current->as, va, pa, 0);
11.    }
12.    current->max_brk = new_brk;
13.  }
14.  return 0;
15.}
```

5.3 基于时间片的进程调度

1. 在 cpu 结构体中添加一个 bool 成员 INTR。

2. 在 dev_raise_intr 中将 INTR 引脚设置为高电平。

3. 在 exec_once 的末尾添加轮询 INTR 引脚的代码，每次执行完一条指令就查看是否有硬件中断到来。

```
1. vaddr_t exec_once(void) {
2.   decinfo.seq_pc = cpu.pc;
3.   isa_exec(&decinfo.seq_pc);
4.   update_pc();
5.   if (isa_query_intr()){
6.    update_pc();
7.   }
8.   return decinfo.seq_pc;
9. }
```

4. isa_query_intr、raise_intr 函数的实现

isa_query_intr 函数查询是否有中断请求。如果 CPU 的中断标志 INTR 和状态寄存器 sstatus 的中断使能位 SIE 都为真，则清除中断标志并调用 raise_intr 函数触发定时器中断，并返回 true 表示有中断发生；否则返回 false。

```
bool isa_query_intr(void) {
  if (cpu.INTR & cpu.sstatus.SIE) {
    cpu.INTR = false;
    raise_intr(IRQ_TIMER, cpu.pc);
    return true;
  }
  return false;
}
```

raise_intr 函数触发一个中断或异常。它首先禁用全局中断，然后将异常程序计数器 sepc 设置为传入的程序计数器 epc，将异常原因 scause 设置为传入的中断号 NO，并将跳转地址设置为中断向量表 stvec 的值。最后，通过调用 decinfo_set_jmp，设置跳转标志为 true。

```
void raise_intr(uint32_t NO, vaddr_t epc) {
  cpu.sstatus.SIE = 0;
  cpu.sepc.val = epc;
  cpu.scause.val = NO;
  decinfo.jmp_pc = cpu.stvec.val;
  decinfo_set_jmp(true);
}
```

5. 在 CTE 中添加时钟中断的支持，将时钟中断打包成 _EVENT_IRQ_TIMER 事件.

Nanos-lite 收到_EVENT_IRQ_TIMER 事件之后，调用_yield()来强制当前进程让出 CPU。

5.4 运行结果

分页机制、进程上下文的切换、时钟驱动的进程上下文的切换基

本实现。可以在时钟驱动的切换下，同时运行两个用户进程——hello和仙剑奇侠传。

6　实验结果与结果分析

本次课程设计涵盖了计算机系统的多个重要方面，包括调试技术、RISC-V 虚拟机开发，以及操作系统构建。

主要成果：

1. 调试器开发

实现了基本的调试功能，如单步运行、断点设置和表达式求值。帮助学生理解程序执行流程，提高代码调试能力。

2. RISC-V 虚拟机 nemu 开发

完成了指令集的实现，添加了 IO、中断和虚存支持。这让学生深入了解 CPU 架构和指令集设计。

3. 操作系统 nanos-lite 开发

实现了进程管理、内存管理和文件系统。引入了分页、进程上下文切换和基于时间片的进程调度。全面展示了操作系统的核心组件和算法。

学习收获：

1. 系统级编程：通过开发调试器和虚拟机，掌握了编程的技巧和挑战。

2. 硬件与软件交互：了解了 CPU 指令与操作系统之间的复杂关系。

3. 操作系统原理：深入学习了进程管理、内存管理、文件系统等关键概念。

4. 并发编程：实践了进程调度和上下文切换，这对理解现代操作系统至关重要。

5. 系统设计思维：从需求分析到实现，培养了全局视野和系统化思维。

总结：

这门课程不仅教授了具体的编程技能，更重要的是培养了系统思维和解决复杂问题的能力。通过参与该项目，我获得了理论知识与实践经验的完美结合，为未来在计算机系统领域的发展奠定了坚实基础。

参考文献