華中科技大學

2018

系统能力综合训练 课程设计报告

题　　目：	X86 模拟器设计
专　　业：	计算机科学与技术
班　　级：	ACM1501
学　　号：	U201514613
姓　　名：	苟桂霖
电　　话：	13006369315
邮　　件：	fluorinedog@gmail.com
完成日期：	2018-1-15 周三凌晨

计算机科学与技术学院

华中科技大学课程设计报告

目录

1 PA0

本节的主要目的为配置环境，搭建好良好易用的基础设施。

PA 原本自带的手动编写的 Makefile 已经不能满足现代编程的需求，因而我们选择将其更换为 CMake, 这样一来我们就可以在 CLion 等 IDE 上进行开发。

与此同时, 为了专注与体系结构, 避免将宝贵的时间浪费于 regex/linked-list 等基础数据结构的实现上，我们将整个项目向 C++ 17 上迁移，以充分利用其丰富强大的标准库与语法糖。

1.1 配置环境

由于我选择使用原生的 ArchLinux 环境, 无需折腾复杂的 Docker 镜像。按照任务书使用 git clone 下载对应仓库后, 运行 init.sh 脚本完成。

1.2 CMake 迁移

首先, 我们需要分析 Makefile 的功能

1. 将 src 下所有 .c 文件编译为 .o 文件
2. 引用 include 下的头文件
3. 加入必要的编译选项
4. 在每次编译时都进行 git commit 操作

前 3 点都有对应的 CMake 命令可以实现。为此，我编写了脚本 cmake_gen.py 自动化这一步骤。至于第四点，对于不熟悉 git 的同学来说可能非常有价值，但是对于我本人没有意义，反而对我使用 git diff 造成了障碍。因此这一点被我无视，同时原版 Makefile 的这一功能也被我注释掉。

在迁移过程中, 我还发现了原版代码需要添加-Wno-restrict 编译选项来避免高版本的 gcc 的报错.

1.3 C++ 迁移

第一步, 当然是将.c 文件全部改成.cpp 文件

```
find . | grep -E "\.c" | xargs rename .c .cpp
```

接下来, 由于 C++ 是一门强类型语言, -fpermissive 被默认启用且在高版本的编译器上不可关闭. 因此, 我们需要对 void* 指针的隐式转换进行相应的修改

```
// in include/common.h
template<typename To>
inline void BITCAST(To& dst, const void* src){
    dst = (To)src;
}

// in some src file
// ...
  // ref_difftest_memcpy_from_dut = dlsym(handle,
"difftest_memcpy_from_dut");
  BITCAST(ref_difftest_memcpy_from_dut, dlsym(handle,
"difftest_memcpy_from_dut"));
// ...
```

C++ 不支持 non-trivial designated initializers, 因此需要在 keyboard.cpp 下做出如下修改

```
// #define XX(k) [name_concat(SDL_SCANCODE_, k)] = name_concat(_KEY_, k),
// static uint32_t keymap[256] = {
//   _KEYS(XX)
// };
static uint32_t keymap[256] = {};
static void init_keymap(){
    #define XX(k) keymap[name_concat(SDL_SCANCODE_, k)] =
name_concat(_KEY_, k);
    _KEYS(XX)
}
```

出于安全因素, C++ 不允许将常量字符串赋值给 `char *`类型, 因此需要添加 `const` 修饰符.

2 PA1

PA1 的主要目的是实现一个基本的 x86 模拟器的基础设施。

2.1 实现正确的 CPU_State 结构

根据 x86 手册，CPU 内的通用寄存器有 8 个，可以根据其位次访问 gpr[i], 也可以使用其别名访问。因此, 我们需要正确实现 union 结构，保证两种访问方式的是等价的。

```
typedef 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;
  };
} CPU_state;
```

2.2 RTFSC

本模拟器的核心在与 decode 和 execute 函数, 他们根据指令的不同, 形态迥异. 目前仅有 mov 被实现了, 其具体细节暂时作为黑箱处理.

1. 程序调用 init_monitor 进行初始化步骤
    a. parse_args 解析命令行参数
    b. init_log 打开 log 文件作为调试输出
    c. reg_test 测试 CPU_State 是否正确实现
    d. load_img 加载镜像, 如果命令行参数没有给出, 加载一个只有 mov 和 halt 指令的小镜像到 ENTRY_START 处
    e. restart 将 EIP 设置为 ENTRY_START, 也就是默认的镜像入口点
    f. init_regex 初始化正则引擎

g. init_wp_pool 初始化 watchpoint 的链表
h. init_device 暂时为空, 等待后期实现 IO_Exception
i. init_difftest 等待后期实现 diff 测试

2. 接下来, 进入了 ui_mainloop, 也就是 NEMU 图形界面的主要消息循环
   a. 在 batch_mode 下, 跳过交互代码, 直接执行 cpu_exec, 不然进入 NEMU 命令行模式, 详解如下:
   b. 使用 readline 读取 NEMU 命令
   c. 在 cmd_table 查找执行对应的 handler
   d. help 打印帮助信息
   e. q 退出
   f. c 执行 cpu_exec
3. 上一步最终都要进入 cpu_exec, 以下详述 cpu_exec
   a. 判断 nemu_state, 在 ABORT/END 等状态终止执行
   b. 设置必要符号后, 切入 exec_wrapper
      i. decoding 为全局状态变量, 首先设置 seq_eip 为 EIP
      ii. 切入 exec_real 中, 分析指令的第一个字节, 设置对应的位宽, opcode, 然后切入 idex 中。分别执行指令的 decode 和 execute 函数, 完成指令的解码执行
      iii. 打印调试信息
      iv. 更新 EIP
   c. 用户程序指令计数器喜+1
   d. 检查 watchpoint
   e. 打印若干调试信息
   f. 循环执行, 直到 NEMU 状态不再是 RUNNING 时退出
   g. 命令行模式读取下一条指令, 或者在 batch 模式下直接退出

2.3 单步执行

解析需要执行的步数, 调用 cpu_exec 即可。

2.4 打印寄存器

仿照 gcc 的格式, 将 8 个寄存器按照 GPR 的顺序输出。

2.5 扫描内存

根据内存首地址, 将连续 N 个 16 进制数输出即可。

2.6 计算表达式

我们采用了伪编译的方法, 将表达式转换为可以直接执行的递归函数二叉树, 以减少 watchpoint 的 Expr 的执行开销.

2.6.1 Tokenizer

我们没有采用框架代码, 而是给予 std::regex 实现了一套 tokenizer 库. 鉴于编译仅需一次, 稍微缓慢的算法是可以接受的, 因此我们将每一条规则都编译成 对象后, 对每一个新的表达式尝试匹配所有规则, 以取出一个 token, 如此重复直到取出所有的 token

2.6.2 算符优先文法

我们支持了大量二元运算符, 并且完全按照 C++的运算符优先顺序和结合性进行求值. 以下为优先级升序排列的符号表

优先级	运算符
1	||
2	&&
3	|
4	&
5	== !=
6	<= >= < >
7	>> <<
8	+ -
9	* / %
10(unary)	* ! ~ - +

为了支持如此多的运算符, 我们采用了算符优先文法作为二元运算的引擎. 比较左右两个二元运算符的优先级, 如果左边的运算符优先级更高, 应该首先执行左边的计算, 表现在代码中就是当前节点作为右儿子合并到左边的子树上. 如果右边的运算符优先级更高, 应该优先执行右边的计算, 表现在代码中, 就是尝试解析右边符号串生成新的子树.

2.6.3 LL(1) 文法

对于单目运算符以及括号的部分, 我们采用进行 LL(1)递归下降进行分析. 编译原理教程有详细的解释, 在此不在赘述。最后, 我们可以得到一棵表达式树, 使用 tree->eval() 即可递归地完成求值操作.

2.7 补充命令行功能

有了表达式引擎后, 我们就可以愉快地补充我们 NEMU 命令行的功能了, 列表如下

help	打印帮助
c	继续执行
q	退出
si [N]	单步执行 N 条指令
info r	打印寄存器状态
info w	打印 watchpoint 状态
p EXPR	计算表达式
x N EXPR	扫描内存
w EXPR	设置 watchpoint
d N	删除 N 号 watchpoint

前 5 个功能已经在以前的阶段中实现, 我们只需要其他功能即可

2.7.1 内存扫描

内存扫描实际上就是按照格式将内存数据打印出来即可. 我们仿照 gdb 的格式, 生成了如下效果

```
(nemu) x 16 $eip
0x00100000: 0x001234b8    0x0027b900    0x01890010    0x0441c766
0x00100010: 0x02bb0001    0x66000000    0x009984c7    0x01ffffe0
0x00100020: 0x0000b800    0x00d60000    0x00000000    0x00000000
0x00100030: 0x00000000    0x00000000    0x00000000    0x00000000
```

2.8 Watchpoint

Watchpoint 在原先的框架中是一个限制了长度的链表. 我们将其删除, 改为 std::map 存放. 因此, 添加删除与遍历 watchpoint 的方法就非常简单了, 只需要调用相应 API 即可.

为了链表有一个全局唯一的编号, 我们定义了一个全局自增变量 g_watch_count 以供使用.

WatchPoint 结构体中存放了伪编译好的表达式, 以及上一次更新时的表达式. 在 CPU_exec 中, 每当执行一条指令, 都需要更新 watchpoint 信息. 如果发现其中有值发生了改变, 需要打印新旧值, 和检视点信息, 并停止执行.

最后效果如下

```
(nemu) w 123
Added to watchpoint 0
(nemu) w 456
Added to watchpoint 1
(nemu) w 789
Added to watchpoint 2
(nemu) info w
id      value           expr
0       123             123
1       456             456
2       789             789
(nemu) d 1
Watchpoint 1 deleted
(nemu) w *($eip)
Added to watchpoint 3
(nemu) c
Software watchpoint 3: *($eip)
Old value = 1193144 [0x001234b8]
New value = 268445625 [0x100027b9]

(nemu) info w
id      value           expr
0       123             123
2       789             789
3       268445625       *($eip)
(nemu)
```

2.9 必答题

2.9.1 理解基础设施

理解基础设施 我们通过一些简单的计算来体会简易调试器的作用. 首先作以下假设:

假设你需要编译 500 次 NEMU 才能完成 PA.

假设这 500 次编译当中, 有 90%的次数是用于调试.

假设你没有实现简易调试器, 只能通过 GDB 对运行在 NEMU 上的客户程序进行调试. 在每一次调试中, 由于 GDB 不能直接观测客户程序, 你需要花费 30 秒的时间来从 GDB 中获取并分析一个信息.

假设你需要获取并分析 20 个信息才能排除一个 bug.

那么这个学期下来, 你将会在调试上花费多少时间?

$$500 * 0.9 * 30 * 20 = 270{,}000s = 75\ hr$$

由于简易调试器可以直接观测客户程序, 假设通过简易调试器只需要花费 10 秒的时间从中获取并分析相同的信息. 那么这个学期下来, 简易调试器可以帮助你节省多少调试的时间?

$$500 * 0.9 * (30 - 10) * 20 = 180{,}000s = 50\ hr$$

2.9.2 查阅 i386 手册

EFLAGS 寄存器中的 CF 位表示运算发生了无符号进位, 需要查阅 ***Vol.1 3.4.3.1 Status Flags.***

ModR/M 字节用来选择寄存器和内存地址/寄存器的各种模式。需要查阅 ***Vol.2A 2.1.3 ModR/M and SIB Bytes*** 以及其指出的扩展阅读页

mov 指令的具体格式可以直接参见 ***Vol2B 4-35 MOV***

2.9.3 shell 命令

使用 cloc 进行统计

```
      87 text files.
      87 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.74  T=3.25 s (25.9 files/s, 1522.5 lines/s)
-------------------------------------------------------------------------------
Language             files          blank        comment           code
-------------------------------------------------------------------------------
C++                     31            479            148           2165
C/C++ Header            24            181             65            717
C                        4            102             76            406
Markdown                 7             67              0            299
make                     3             31             16             71
```

```
Python                   2             7             3            43
Bourne Shell             1             6             0            34
CMake                   12             0             0            29
-------------------------------------------------------------------------------
SUM:                    84           873           308          3764
-------------------------------------------------------------------------------
```

使用 git diff –stat <old> <new>

```
48 files changed, 1362 insertions(+), 485 deletions(-)
```

2.9.4 Makefile

-Wall -Werror 将开启编译器所有警告，并将警告视作错误，阻止编译，保证代码中 no warnings，以避免 bug

2.10 思考题

- 假设你在 WINDOWS 中使用 DOCKER 安装了一个 GNU/LINUX CONTAINER, 然后在 CONTAINER 中完成 PA, 通过 NEMU 运行 HELLO WORLD 程序. 在这样的情况下, 尝试画出相应的层次图.

层次图
"Hello World" program
Simulated x86 hardware
NEMU
Docker
Windows
Computer hardware

- 如果没有寄存器, 计算机还可以工作吗? 如果可以, 这会对硬件提供的编程模型有什么影响呢?

寄存器是计算机储存体系结构的最顶层. 如果没有寄存器, 那么所有的信息都必须来自内存. 所有的指令都必须直接在内存上运行. 由于内存空间很大, 所有的指令可能都需要带上很长的位段, 描述从内存何处取值, 或者划分内存的一段极小的区域作为"指令高速区域"进行操作, 但是前者会极大增大指令体积, 后者实际上和寄存器方法没有太大区别.

- 我们知道, 时序逻辑电路里面有"状态"的概念. 那么, 对于 TRM 来说, 是不是也有这样的概念呢? 具体地, 什么东西表征了 TRM 的状态? 在状态模型中, 执行指令和执行程序, 其本质分别是什么?

其状态包含存储器, PC, 寄存器的值. 执行指令, 等价于以储存器中的值作为输入, 依照对应的指令进行一次状态的变换. 执行程序就是这一系列变换过程的总和.

- 回忆程序设计课的内容, 一个程序从哪里开始执行呢?

程序拥有 EFI 结构体, 其中有个 entry_point 参数决定了程序执行的入口地址.

- 阅读 reg_test()的代码, 思考代码中的 assert()条件是根据什么写出来的.

第一部分, 是根据 x86 的小端序结构, 8bit 读取被写入的 32bit 得到了低字节

第二部分测试别名. 保证 aliasing 的变量等价

- 在 cmd_c()函数中, 调用 cpu_exec()的时候传入了参数-1, 你知道这是什么意思吗?

cpu_exec 的参数表示指令执行的最大条数, 其类型是 uint64, 输入-1 被转换为 UINT64_MAX, 表示无限执行下去直到 halt

- "调用 cpu_exec()的时候传入了参数-1", 这一做法属于未定义行为吗? 请查阅 C11 手册确认你的想法.

根据手册 6.3.1.3, 这是安全的.

```
1 When a value with integer type is converted to another integer type
other than _Bool, if
the value can be represented by the new type, it is unchanged.
2 Otherwise, if the new type is unsigned, the value is converted by
repeatedly adding or
subtracting one more than the maximum value that can be represented in
the new type
until the value is in the range of the new type.
```

- opcode_table 到底是个什么类型的数组?

它是以结构体 opcode_entry 为元素的数组, opcode_entry 包含了解码函数, 执行函数, 指令位宽.

- 但你是否怀疑过, 凭什么程序执行到 main()函数的返回处就结束了?

main 函数的返回想当于调用 exit 函数, 会执行一下清理工作, 比如已经被 atexit 注册好的函数

- 对于 GNU/Linux 上的一个程序, 怎么样才算开始? 怎么样才算是结束? 对于在 NEMU 中运行的程序, 问题的答案又是什么呢? 与此相关的问题还有: NEMU 中为什么要有 nemu_trap? 为什么要有 monitor?

在操作系统中运行的程序，调用 syscall_exit 算结束。而在 nemu 中，nemu_trap 相当于停机指令

- 如何测试字符串处理函数?

可以随机你生成若干个字符串, 用 1 到多个空格分隔开来. 使用 args 解析后, 能够恢复原数组即可.

- 实现带有负数的表达式求值

使用 LL(1) 文法即可.

3 PA2

本节需要实现一个基本可以使用的 NEMU 模拟器

3.1 RTFSC

在实现 dummy 的过程中, 我们阅读了源代码, 有以下一些值得注意的要点:

1. 每一条指令都分为 解码阶段和执行阶段
2. 解码阶段按照指令格式, 可以处理多种指令格式的排列组合, 包括
    a. opcode 低位编码
        - ModR/M
        - imm8/imm16/imm32
        - 这些格式排列组合后, 可以表示出
        - 通用寄存器
        - 内存地址
        - 立即数
        - debug/control register
        - eflags
        - j 系列指令的相对寻址
        - ds:si/es:ds
    b. 解码函数完成后, 其值被临时储存于 id_src, id_dest, id_src2 中
        - 读取时, 应该采用 id_xxx->val 访问相应的值
        - 写入时, 应该利用 operand_write 函数
    c. 执行函数按照如上的约定存取数据, 改写 EFLAGS 等.
3. opcode_table 存储了指令解码函数(可以为空)和执行函数的查找表. idex_wrapper 读取相应的函数, 分别执行
4. 所有的解码和执行函数都使用 rtl 伪指令作为数据流传递的方法.

5. 为了保证后期可以成功完成各种 JIT 的目标, 同时尽可能得减少 BUG 出现的概率, 我们对 RTL 伪指令的书写做出以下约定:
    a. 所有的 RTL 伪指令不允许改变任何寄存器状态, 如果需要临时寄存器, 应该在函数定义域中临时声明.
    b. 所有伪指令都必须由基本 RTL 伪指令或其他伪指令组合而成, 不允许递归
    c. 不允许直接读取 rtlreg_t 中的数据, 只能借助 RTL 伪指令进行处理

在阅读文档的过程中，我和原作者进行了沟通交流，我了解到所有的指令都应该直接或者间接使用 RTL 函数实现，以方便后续 JIT 的实现。

原作者认为，所有的临时变量都应该使用固定的全局静态变量，而经过我的仔细思考，我认为这个需求不合理，因此我大量使用了函数内部的临时变量，极大增大了可读性，降低了 BUG 出现的可能性。

3.2 EFLAGS

我们需要优雅地添加对 EFLAGS 的支持. 目前, 我们只需要支持 IF/SF/ZF/OF/CF. 为了减少代码的冗余度, 我们采用了宏模板的黑魔法, 在 eflags.h 中定义了 EFlags 中的所有位段, 并添加

- mask 某个 flag 的相应位掩码
- offset: 某个 flag 的在 eflags 中的偏
- lowmask: mask 右移 offset 到最低位的值, 配合 offset 使用可以提取相应的位.

3.3 运行第一个客户程序

为了实现 dummy，需要实现 call, push, sub, xor, pop, ret 六条指令。这些指令实现成功后, 可以得到正确输出。

3.4 添加 diff-test 支持

在开启 DIFF_TEST 宏后运行, 发现程序 assert 停止在 diff_test 处的 TODO 上. 对于通用寄存器, 框架已经基本搭好, 只需要在在 diff-test.cpp 处添加比较语句即可

对于 EFlags, 比较麻烦, 除了在 qemu/nemu 直接拷贝并比较数据外，还需要额外在设计一套架构，指定何时可以忽略 eflags 的某些位（如 shift 类指令对 OF 的实现是未定义的，必须进行忽略处理）

3.5 实现更多指令

因此, 每一条指令的实现, 分为以下几个步骤:

- 找到对应的解码函数, 如果没有, 实现它
- 实现对应的执行函数, 并在 all-intr.h 中声明它
- 在 exec.cpp 文件里的 optable_table 上注册相应的函数

指令的具体实现方式查看 intel 手册即可。

3.6 实现字符串处理函数

直接照搬<cstdlib>的做法即可。完成后，strings 即可正确通过测试。

3.7 实现字符串打印

我们在 nexus-am/libs/klib/src/stdio.c 下，模仿 C++的类实现了一套 struct OutputEngine 和对应的 handler，将 printf/snprintf/sprintf 进行了统一处理，仅有边界条件，字符输出函数，目标地址不同。

对于 Xprintf 的 format 串，我们实现了%d, %s, %p, %x 功能，以及简单的控制字符和占位符处理，方便后续 debug 进行 log 输出

完成后，hello-str 可正常通过测试。

3.8 添加 io

实现 in/out 支持即可。

值得注意的事，in/out 暂时没有对应的 RTL 指令，因此可能在未来对 jit 产生影响，所以加上//JIT_TODO 的注释

3.9 实现 IOE 和 MMIO

实现时钟，只需要处理_DEVREG_TIMER_UPTIME 即可，使用 unix time 获取时间后填写。

对于键盘，处理_DEVREG_INPUT_KBD 即可

而针对 VGA，首先用同样方法处理_DEVREG_VIDEO_INFO, 然后利用设计好的 MMIO 接口，添加对_DEVREG_VIDEO_FBCTL 的处理。

3.10 Microbench

关闭 debug 与 difftest，使用-DCMAKE_BUILD_TYPE=Release 进行编译，得到的 microbench 跑分。PA5 实现了 JIT 并进行了大量优化后，得分可高达 2038.

3.11 展示系统

玩打字游戏，勉强能维持几帧的 FPS

放 ppt，播放较为流畅

运行 litenes，玩 Mario，可以勉强运行，但是卡成了 ppt.

3.12 必答题

3.12.1 编译与链接

由于在 PA0 中，我将 nemu 的整套框架迁移到了 C++上，因此 inline 和 static inline 等价，都可以正常通过编译链接，程序等价。

仅保留 static，然后进行 objdump，发现其多了一部分 interpret_rtl_li 符号，地址也不相同，这是因为每个翻译单元都包含一份此函数的弱符号

去掉 static inline，出现大量 multiple definition of `interpret_rtl_li，因为每个翻译单元包含此函数的强符号，因此无法通过编译。

3.12.2 编译与链接

1. 在 nemu/include/common.h 中添加一行 volatile static int dummy; objdump -t nemu | grep -w ZL5dummy | wc -l 有 29 个 dummy

2. 添加上题中的代码后, 再在 nemu/include/debug.h 中添加一行 volatile static int dummy; 编译无法通过，因为 c++禁止重复定义变量，哪怕时未初始化的变量。 C 版本有 30 个，因为允许重复定义，看作一个。
3. 初始化后，也无法通过编译。

3.12.3 了解 Makefile

由于我将整套 Makefile 的逻辑迁移到了 CMake 工程下，我在 PA0 就已经认真研究了 Makefile 构建系统的逻辑。

观察发现，其套路和大多数 Makefile 工程构建流程一致

1. 找出所有.c 文件
2. 使用 gcc 找出.c 包含的头文件，生成.d 文件
3. 每一个.c 编译为目标文件.o 时，同时依赖.d 中的文件，以便未来重新构建
4. 将所有.o 链接起来形成最终的可执行文件
5. 最后添加.PHONY 命令 app, run, gdb, clean 等

3.13 思考题

3.13.1 捕捉死循环

这是不可计算问题。捕捉所有死循环等价于解决图灵机停机问题。即使是对于有限内存的模型，也是一个计算复杂度不可接受的 NP 问题。但是，我们依然可以捕捉部分死循环：在 jcc 语句处，记录机器当前的<寄存器，内存>状态，如果发现和历史状态相同，则可以判断出现了死循环。

3.14 主要故障

3.14.1 加入 eflags ignore 后，依次出现部分指令 eflag 出错的问题

发现虽然在 eflags 未定义的指令加入 ignore 后跳过了该指令的检查，但是如果其后紧接不改变 eflag 的指令，如 mov，再次进行检查依然会报错。解决方法是保持上一条指令的 eflags ignore 状态，直到 update_XF 时更新

4 PA3

4.1 实现中断机制

CPU 处添加 idtr 寄存器，包含 limit 和 base

在 NEMU 中实现 lidt/int 指令，其中 int 使用了 raise_intr 函数

注意到，我们认为 CS 永远为 8，以保持和 QEMU 的一致性

4.2 _Context 结构体设计

根据 trap.S 汇编指令以及 pusha/int 的 RTL 指令细节，按顺序填写_Context 的结构体定义即可。注意到 call irq_handle 之前的 push 指令提供了 irq_handle 的第一形参。

4.3 事件分发

保存上下文后，调用 irq_handle 进行真正的中断处理。Irq_handler 根据中断号设置事件类型，调用 user _handler，也就是操作系统的 do_event 进行后续处理。

4.4 恢复上下文

实现 pusha/popa/iret 即可

4.5 实现 loader

在 naïve_load 中，将 ramdisk 位置的数据拷贝到 ENTRY_POINT 处即可

4.6 识别系统调用

在 irq_handle 与 do_event 上加入 0x80 的识别，调用 do_syscall, 分别实现 SYS_exit, SYS_yield, SYS_write, SYS_brk

实现 SYS_exit, 直接调用开机菜单程序选择

SYS_yield 仅打印一次 log

SYS_write 使用_putc 写串口。此时，hello 程序运行时，每一个字符会调用一次这个 syscall

SYS_brk 返回 0 即可，hello 此时每行打印才会调用一次 SYS_write 对应的 syscall

4.7 文件系统与 VFS

首先实现通用的 VFS 接口：vfs_read/vfs_write/vfs_lseek/vfs_open/vfs_close，在 do_syscall 中匹配相应的系统调用。

对于真实文件，为 ramdisk_xxx 系列的函数调用适配 VFS 接口，在初始化过程 init_fs 中，初始化所有真实文件 read/write 回调函数为 ramdisk_read/ramdisk_write

对于串口，为 stderr, stdout, /dev/tty 适配 serial_write

对于 VGA，为/dev/dispinfo 适配 dispinfo_read, 首先预处理时填写相应的图像显示信息。对于/dev/fb，适配使用了 draw_rect 的 fb_write（后期为了图像显示的性能，我们绕开了 draw_rect, 直接使用了 draw_direct）

对于事件，实现 events_read 对键盘和时钟的处理即可。

4.8 自由开关 diff-test

添加 attach 和 detact 命令，在 attach 时打开控制 diff-test 开关的 bool 变量，将指定区域的内存复制到 qemu 中，并执行一段代码设置 idtr，最后将 cpu 状态复制过去

关闭时，设置 bool 变量为 false 即可。

4.9 实现快照

save 命令中，保持 cpu 状态和内存信息，同时保存内存映射 mmio 中的数据使得图像输出也能够复原。

load 按保存顺序加载即可。

4.10 添加开机菜单，展现系统，运行 Mario

实现 SYS_execve，以切换程序。为了 litenes 的正常运行，为其增加了参数传入功能。

进入 init 后，按照菜单选择选择运行各种程序。

4.11 必答题

4.11.1 文件读写的具体过程

对于 fread

1. fread 是 libc 中的 C 运行时库函数，调用 libos 的_read 进行读
2. _read 是 SYS_read 的封装，直接使用系统调用 int 切入操作系统
3. 通过中断向量表进入 vecsys，通过 trap 和 irq_handler，最终抵达 do_event 中进行事件分发，进入 do_syscall
4. Syscall 调用 vfs_read 抽象层处理文件读，回调进入 ramdisk_read
5. Ramdisk 实际完成了以上工作，得到返回值后逐级返回
6. Nemu 执行以上功能的指令

对于 redraw

1. NDL_DrawRect 使用 NDL_Render 绘制图像，首先保存到 canvas 中
2. NDL_Render 使用 fread/fwrite 向/dev/fb 写入 canvas 数据, 最后使用 fflush 刷新
3. fseek/fwrite 配合使用，将数据暂时写入 C 运行时的缓冲区的对应位置
4. fflush 分别使用 libos 的_lseek/_write 将数据写入/dev/fb 中
5. _lseek/_write 是相应 syscall 的简单封装，使用 int 切入操作系统
6. 分发后直达 vfs_lseek/vfs_write
7. 对于 vfs_lseek, 调整操作系统文件指针即可
8. 对于 vfs_write， 回调进入 fb_write
9. fb_write 使用 draw_direct 直接将数据写入 VMEM 地址范围
10. nemu 的 paddr_write 检查地址落入了 mmio 的范围，调用 mmio_write
11. mmio_write 的将数据写入 mmio_space_pool
12. 在 device_update 中，调用 SDL 将图像更新到屏幕上

4.12 思考题

4.12.1 对比异常处理和函数调用

异常处理时，所有的寄存器状态都需要保存，而函数调用 callee-saved register 和 eflags 无需保存。

4.12.2 诡异的代码

push %esp 实际上为 irq_handle 提供了第一形参，指向_Context

4.13 主要故障

4.13.1 brk 在 native 下不可以工作

询问同学实现，发现自己对 brk 的工作原理理解有问题。在 nemu 和 nanos 中实现的两套错误实现虽然可以工作，但是和 native 的正确实现无法对接，因此推翻重写。

5 PA4

5.1 实现 kernel 函数的上下文切换

1. 实现 CTE 的_kcontext()函数，在栈底构建_Context 信息。并在栈顶放置指向它的指针
2. schedule()函数，返回&pcb[0]
3. do_event 收到_EVENT_YIELD 事件后, 调用 schedule()并返回其现场
4. 修改 CTE 中 asm_trap()的实现, 使得从 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
5. 测试 hello_fun，成功完成

5.2 实现用户函数的上下文切换

1. 实现_ucontext, 在栈底依次构建 argv 指针数组，argv 指向的内容，argc/argv/envp，以及_Context，最后在栈顶放置指向它的指针
2. _switch 中设置_Protect 指针，在_irq_handle 中加入对_Protect 的处理
3. 测试/bin/pal 可运行

5.3 实现分页机制

1. 首先，给 cpu 加入 cr0, cr3 寄存器，并实现 mov debug 指令。
2. 定义 HAS_VME，实现 cr3 到 page table 再到 page entry 的转换，分别在 nemu 和 nanos-lite 的对应位置按照 intel 手册填写转换功能。
3. 注意到 vaddr_xxx 处需要对跨页访问进行特殊处理
4. 在_vme_init 和_ucontext 中，加入_Protect 的处理，也就是用户程序的 cr3 保存机制。
5. 在_switch，irq_handle 加入 cr3 的切换
6. 实现_map 填写页表项，同时自动生成 PageTable

5.4 分页上运行用户程序

将直接复制 ramdisk 改为一页一页加载，同时修改默认加载地址以避免冲突

5.5 用户程序虚存管理

修改_sbrk 实现，当其需要的 brk 超过当前的 max_brk 时，分配新的页以满足需求。

5.6 支持开机菜单

添加对 execve 的支持，主要是重新加载新的页表

5.7 实现时钟中断

增加 intr 引脚和 IF 的处理，添加 0x20 中断描述符，trap.S 添加 vecirq，在 irq_handler, do_event 加入对应的识别与处理，在 nemu 中 exec 下添加时钟中断的查看功能。

5.8 展示计算机系统

使用 kload 设置 pcb[0]为前台进程，使用 uload 设置 pcb[1], pcb[2], pcb[3]为后台进程。

在文件系统中给每一个进程单独设置独立的进程指针，以支持并发读写。

在 events_read 中加入 F1, F2, F3 的检测，切换进程号，在 schedule 中真正切换进程

Schedule 中设置前台进程为较小的时间比例，后台较大，以提高整体性能

5.9 必答题

5.9.1 分页机制和硬件中断时如何支撑分时运行的

1. 初始化过程中，内存管理，物理文件系统，外部 IOE，IRQ，虚拟文件系统，进程被依次初始化。
2. 内存管理初始化过程中，创建了恒等映射的内核页表，将 cr3 设置为内核页表项，并开启分页机制。
3. irq 初始化了中断向量表，设置了中断寄存器
4. 进程块使用 kload 和 uload 分别加载了 hello 和 pal，其中 hello 复用了内核页表项，而 pal 创建了自己新的页表项
5. uload 同时还创建了 pal 加载的页表项，指向包含 pal 数据虚拟地址空间。而 kload 只是复用了内核页表。
6. 用户进程创建完毕后，启用 schedule 进行第一次调度

7. 调用_yield，切入 schedule 中
8. schedule 有千分之一的概率加载 hello，此时，hello 的页表项被加载进 cr3 中，_Context 的地址也被指向 hello 的对应地址。irq_handler 返回后，汇编代码恢复上下文。iret 后，eip 转到 hello_fun 的函数入口开始指向，直到_yield 重新 schedule
9. 其余情况下，schedule 选择加载 pal，此时，pal 的页表项被加载进 cr3 中，_Context 的地址也被指向 pal 的对应地址。irq_handler 返回后，汇编代码恢复上下文。iret 后，eip 转到 ENTRY_POINT，cr3 将其映射到包含 pal 程序与数据的虚拟空间中，开始游戏的执行过程。
10. 硬件中断信号由外部设备给出，nemu 执行一条指令就会检查时钟引脚和 IF 位是否满足中断条件，如果满足，就调用 raise_intr 启动时钟中断，和 int 流程基本一致: 中断向量表查找到 vecirq，asm_trap 保存上下文和中断号，irq_handler 解析 event 类型，do_event 分发处理，直到调用了_yield，然后重复 7,8,9 的处理操作。

5.10 主要故障

5.10.1 切换进程时触发文件系统范围限制 assert

文件系统一开始在整个系统范围类只有一个文件指针，多进程 lseek/write 会得到错误的行为。需要为每个进程单独维护文件指针已解决并发问题。

5.10.2 程序莫名运行到错误地址。

通过查看执行的程序汇编码，发现和 objdump 给出的汇编不一致。给虚拟地址的 text 段加上写保护（assert），发现在操作系统处触发，进一步检查发现是一个指针被设置为了错误的地址，在用户程序 text 段写入了错误的数据。

5.10.3 程序字体不显示

通过查看执行的程序汇编码，又发现和 objdump 给出的汇编不一致。发现地址恰好接近 4k 页的尾部，查看 vaddr_read 发现没有对跨页读进行处理。

6 PA5

6.1 浮点数支持

原框架已经支持

6.2 JIT 技术

我们选用 LLVM ORC v2 框架进行 jit 编译。

6.2.1 映射关系

原来框架使用了“覆写“技术，也就是每一个临时变量对应一个内存地址，rtl 也就是对内存中的数据进行操作，赋值时，数据直接覆盖内存地址的数据。

我为了效率，弃用了这一策略，改为使用 rebinding 技术。如此一来，内存地址仅仅作为一个 tag，类似于 Python 变量名，实际它们指向了一个只读的 Value 对象。赋值时，实际上是创建了新的 Value 对象，让 tag 指向它。如此一来，在翻译时，所有的 Value 可以轻松地在 llvm 后端解决其依赖关系，进行更好的寄存器分配，最后生成的汇编码大多直接使用了寄存器作为中间数据存放地点。

对于 cpu 内部的 rtlreg 变量，我在第一次接触时读取它，而写入时当作普通临时变量做类似处理。当 BuildingBlock 编译结束时，我才将 dirty 的 cpu 变量写回 cpu 结构体中，极大减少了内存开销。

6.2.2 编译策略

我按照程序块进行编译。在编译过程中，由于程序块还没有完成，只能解释执行。编译完成后，代码以函数指针的形式存放于 icache 中，如果遇到相同的 eip 就直接调用此函数代替解释执行。

6.2.3 性能优化

裸的 jit 完成后，microbench 达到了 1300+的分数，Mario 性能提升到了中等卡顿水平。分析发现，主要瓶颈在于 vaddr 读写调用函数上。我进一步优化时，做出了一些大胆的限定：

1. 除了代码读以外，不会出现非对齐的内存访问

2. 同一条内存访问指令只能走直接访存、虚拟访存、MMIO 这三条路径中的一种。

基于这些假设，我将 vaddr 拆分，以第一次执行的内存访问路径进行相应的 dispatch. 完成后，microbench 达到了 2000+的分数，Mario 仅有轻微卡顿，PAL 体验和 native 相差无几，证明我的优化思路时正确的。

7 总结

7.1 课设总结

这次试验加深了我对 x86 体系结构的理解，训练了我调试大型程序能力，积累了构造复杂工程的经验。JIT 选做部分让我对程序编译优化有了初步感知。

7.2 心得

1. 善于使用 assert 和 panic 可以大幅减少 debug 的时间
2. 写指令的时候，一定要先参考文档，想当然地写会被一些反直觉的行为坑得很惨
3. 在 RTL 指令书写时，我选用了临时变量，极大的减少了 bug 的出现概率
4. 写代码时如果出现重复功能，尽量不要复制粘贴，而是选用函数/宏等形式复用代码，减少出错概率。

华 中 科 技 大 学 课 程 设 计 报 告

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字: 

二、对课程设计的学术评语（教师填写）

三、对课程设计的评分（教师填写）

评分项目 (分值)	报告撰写 (30 分)	课设过程 (70 分)	最终评定 (100 分)
得分			

指导教师签字: _____