

PA2 实验报告

赵

超懿 匡亚明学院 191870271

我已完成PA2 全部必做内容以及选做malloc实现并运行超级玛丽

必答题：

1.理解YEMU如何让执行程序

执行加法程序的状态机：

在执行加法指令时：不会改变内存的状态，只改变pc和寄存器的值。

$(pc, R[0], R[1], R[2], R[3], \text{内存}) \rightarrow (pc+1, R[0]+R[1], R[1], R[2], R[3], \text{内存})$

pc加1指pc移动到下一个字节。初始 $R[0], R[1], R[2], R[3]$ 的四个位置分别代表寄存器0, 1, 2, 3，图中假设加法指令 $rt=0, rs=1$ ，则 $R[0]$ 位置上的指变为 $R[0]+R[1]$ 。若使用 rs 核 rt 来表示，即 rt 位置上寄存器的值变为 $R[rt]+R[rs]$ 。加法指令不会改变内存上的值，故内存不会变化。

YEMU执行指令的过程：首先从存储的程序中取得指令，然后根据前4为来判断指令类型，然后对操作数进行译码，执行对应的操作，同时在译码过程中，pc指向下一条指令。

将程序看作状态机，其下一个状态既和当前的状态有关，也和输入即指令有关。加法指令是YEMU执行过程中一个特例。

比如mov指令 `MOV reg32, imm32`，其操作码为`0xb8+rd`，后跟有`+rd`，代表后面指令的译码结果是32位的寄存器,并根据`rd`得到目标寄存器，然后会有4字节的立即数。在nemu中，在`exec_once`中首先获取opcode位`0xb8`，switch语句定位到`IDEX(0xb8+rd, ov_I2r, mov)`，然后根据宏定义，先设置位宽位4字节（没有`0x66`开头的情况下），然后调用`def_DHelper(mov_I2r)`进行译码，`mov_I2r`的译码先调用`decode_op_r`译码得到寄存器的编号，然后处理好指针`ddest`指向目标寄存器。然后调用`decode_op_i`译码得到4位立即数，即第二个操作数，处理好指针`idsrc1`，然后为执行部分，调用`def_EHelper (mov)`实现指令。最后返回更新pc，进入下一次循环。

3.程序如何运行：理解打字小游戏如何运行。

答：main开始先进行ioe和video的初始化。

`ioe_init`检测设备是否实现，`video_init`对屏幕初始化，背景设为紫色，对字母设置颜色。在循环中，根据fps确定刷新频率，`game_logic_updata(current)`对现存的所有字符的位置进行刷新，刷新的新位置由当前位置核速度计算得到，对于超出的屏幕的字符，统计错的字符数量，使其停留一段时间。对于时间已到的字符，令其消失。再有一个while循环获取键盘的输入，判断是否退出游戏还是不改变还是使字符消失，`check_hit`函数，判断是否输入正确，若正确成功数量增加，使字符反向返回。最后的render对屏幕先刷新，再对字符的现在的位置进行染色，输出到端口，打印当前的正确核错误信息。

在整个运行过程中，打字小游戏在以x86为架构的nemu运行，使用x86架构的nemu计算出屏幕上每个位置的颜色，然后使用am的软件接口把数据输入到端口，然后nemu从接口获取信息将输出反馈在屏幕上

4.在 `nemu/include/rtl/rtl.h` 中, 你会看到由 `static inline` 开头定义的各种RTL指令函数. 选择其中一个函数, 分别尝试去掉 `static`, 去掉 `inline` 或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

`static inline`都去掉：编译报错，原因在于重复定义，因为在`exec.h`中使用`include`包含了`rtl.h`,会使得同一个函数被多次定义（既在`exec.h`中，也在`rtl.h`中）

去掉`static`产生的结果也是重复定义函数，原因和上面的相同。

只去掉`inline` 编译器会发出警告，提示`function defined but not used`，但是由于`Makefile`中 `-Wall` 和 `-Werror`，会报错。在`decode.c`编译时会定义而不被使用。原因在于`inline`会把函数放到调用者内，而不是把函数放置等待调用，故不会出现函数被定义而不被使用的情况。

编译与链接

1. 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy`; 然后重新编译NEMU. 请问重新编译后的NEMU含有多少个 `dummy` 变量的实体? 你是如何得到这个结果的?

27个，在编译后得到的文件中使用`grep`搜索`dummy`的.o文件，共27个；

`shell grep -r -e 'dummy' |wc -l`，这样得到是文件中含有`dummy`的文件的数量，如果在`grep`后加上`-c`，可以得到总数为54,但是由于`dummy`这样的定义为弱符号，每个文件中最后会随机挑选一个编译，故最终数量仍然为27个。

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`; 然后重新编译NEMU. 请问此时的NEMU含有多少个 `dummy` 变量的实体? 与上题中 `dummy` 变量实体数目进行比较, 并解释本题的结果.

27个，和第一问结果相同。原因在于`common.h`中包含了`debug.h`，所以当`common.h`被`include`时，`debug.h`也被`include`。

3. 修改添加的代码, 为两处 `dummy` 变量进行初始化: `volatile static int dummy = 0`; 然后重新编译NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)

报错：重复定义。原因在于`dug.h`包含在`common.h`，这样会对`dummy`进行两次定义（初始化后即为强符号），所以报错。

了解Makefile 请描述你在 `nemu/` 目录下敲入 `make` 后, `make` 程序如何组织.c和.h文件, 最终生成可执行文件 `nemu/build/$ISA-nemu`.

答：格式 `file : file 1,file 2,... file 3`表示文件`file`由后面的文件构成,省略`file`文件本身

然后省略了`common.h`（不直接`include`的），以及`common.h` `include`的`debug.h` 和`macro.h`.

在执行编译时,起作用的主要是这里

```
$(OBJ_DIR)/%.o: src/%.c
```

```
@echo + CC $<
```

```
@mkdir -p $(dir $@)
```

```
@$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
```

一个.o文件由对应的.c文件及CFLAGS中的编译选项完成

```
INCLUDES = $(addprefix -I, $(INC_DIR))
```

```
CFLAGS += -O2 -MMD -Wall -Werror -ggdb3 $(INCLUDES) \
```

```
-D__ENGINE_$(ENGINE)__ \
```

```
-D__ISA__=$(ISA) -D__ISA_$(ISA)__ -D__ISA_H__=\"isa/$(ISA).h\"
```

可以看到 \$(INCLUDES)给出了要生成.o文件需要的头文件的路径（标准库中的不需要写出），这样是使用gcc的过程中，编译器就可以找到对应的头文件，

```
gcc g-02 -MMD -Wall -Werror -ggdb3 -I./include -I./src/engine/interpreter g-  
gdgd -c -o build/obj-x86-interpreter/device/alarm.o */*.c 这样的格式，便可以生  
成.o文件，最后是文件的链接；
```

```
gcc -O2 -rdynamic -o build/x86-nemu-interpreter
```

```
build/obj-x86-interpreter/device/alarm.o
```

```
build/obj-x86-interpreter/device/audio.o
```

```
.....(省略)
```

```
build/obj-x86-interpreter/engine/interpreter/init.o -lSDL2 -lreadline -ldl
```

链接过程将所有文件链接起来。

makefile中还提供gdb，和run操作，分别对应shell中的gdb和执行二进制文件

整个过程中，gcc -I -I./include -I./src/engine/interpreter,从这两个路径搜寻.h文件，以下是详细包含（直接include的，间接包含的会在-I后的路径去寻找）的文件

alarm.c : common.h time.h signal.h

audio.c: 未实现的情况下为 common.h

serial.c : map.h

keyboard.c: map.h monitor.h SDL.h

intr.c: isa.h

device.c: common.h

vga.c: comon.h

map.c: isa.h paddr.h vaddr.h map.h

mmio.c: map.h

port-io.c: map.h

monitor.c: isa.h paddr.h monitor.h getopt.h stdlib.h

log.c: common.h stdarg.h

ui.c: isa.h expr.h watchpoint.h stdlib.h readline.h history.h monitor.h paddr.h

expr.c: isa.h regex.h paddr.h

watchpoint.c: watchpoint.h expr.h malloc.h

cpu-exec.c: isa.h monitor.h difftest.h stdlib.h time.h watchpoint.h

dut.c: dlfcn.h isa.h paddr.h monitor.h

paddr.c: isa.h paddr.h vaddr.h map.h stdlib.h time.h

main.c: stdio.h stdlib.h string.h

reg.c: isa.h stdlib.h time.h reg.h

decode.c: exec.h rtl.h reg.h decode.h

logo.c: 无

special.c: exec.h monitor.h difftest.h

exec.c : exec.h decode.h

intr.c: exec.h rtl.h

mmu.c ; isa.g vaddr.h paddr.h