PA2实验报告

匡亚明学院 蔡其志

151242002

实验进度

我完成了所有内容

必答题

编译与链接1

在 `nemu/include/cpu/helper.h` 中, 你会看到由 `static inline` 开头定义的 `instr_fetch()` 函数和 `idex()` 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, 你会看到发生错误. 请分别解释为什么会发生这些错误? 你有办法证明你的想法吗?

- 去掉`inline`

  `error: ‘instr_fetch’ defined but not used [-Werror=unused-function]`

  原因：每个模块都把 `instr_fetch()` include了一遍但不是都调用了，由于编译开关打开了`-Wall` 和`-Werror`因此会报出error.

  验证：把`-Wall` 和`-Werror`删掉

- 去掉`static`

  `multiple definition of  'instr_fetch'`
  obj/nemu/cpu/decode/decode.o:/home/FlagC/ics2016/nemu/include/cpu/helper.h:11: first defined here

  原因：没有使用`static`导致每个模块里的`instr_fetch`都是全局变量，因此出现了符号重定义。

- 都去掉

  和去掉`static`一样

  原因：先发现了符号重定义的错误

编译与链接2

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?

   答：112个

   查看符号表，使用`readelf -s obj/nemu/nemu | sed -n '/\<dummy\>/p'`查看所有的`dummy` ，使用`readelf -s obj/nemu/nemu | sed -n '/\<dummy\>/p' | wc -l`输出个数即可

---

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.

   答：112个，与上题相同

   因为一个模块会include `debug.h` 和 `common.h` ，并且两处的 `dummy` 均未初始化。这样由于 `tentative declation` 的原因， `dummy` 只会定义一次，其他地方的 `dummy` 自动成为声明，因此与上题相同。

   关于 `tentative declation`:

> A tentative definition is any external data declaration that has no storage class specifier and no initializer. A tentative definition becomes a full definition if the end of the translation unit is reached and no definition has appeared with an initializer for the identifier

---

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.)

   答： `error: redefinition of ‘dummy’`

   因为我们对两个dummy都进行了初始化，因此不再有 `tentative declation` 了，这样同一符号就会被定义多次，自然会触发重定义error。

   如果我们把二者中的任意一个初始化删去就不会再出错了。

了解**Makefile**

请描述你在工程目录下敲入 `make` 后, `make` 程序如何组织.c和.h文件, 最终生成可执行文件 `obj/nemu/nemu` . (这个问题包括两个方面: `Makefile` 的工作方式和编译链接的过程.)

关于 `Makefile` 工作方式的提示:

- `Makefile` 中使用了变量, 函数, 包含文件等特性
- `Makefile` 运用并重写了一些implicit rules
- 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
- RTFM

答：

- Makefile的工作方式：
  1. make会在当前目录下找名字叫“Makefile”或“makefile”的文件。
  2. 如果找到，它会找文件中的第一个目标文件，并把这个文件作为最终的目标文件。
  3. 如果目标文件不存在，或是目标所依赖的后面的.o 文件的文件修改时间要比目标文件新，那么，他就会执行后面所定义的命令来生成目标文件。
  4. 如果目标文件所依赖的.o文件也不存在或较旧，那么make会在当前文件中找目标为.o文件的依赖性，如果找到则再根据规则生成.o文件。
- 编译链接的过程

  依赖关系：

  `all->nemu->nemu_BIN->nemu_OBJS->*.c,*.S`

  编译：

  具体规则看 `config/Makefile.build` ,这里的 `$(1)` 就是 `nemu`

```
define make_command
 @echo + $(3)
 @mkdir -p $(@D)
 @$(1) -o $@ $(4) $(2)
 endef

 # prototype: make_common_rules(target, cflags_extra)
 define make_common_rules
 $(1)_SRC_DIR := $(1)/src
 $(1)_INC_DIR := $(1)/include
 $(1)_OBJ_DIR := obj/$(1)
# .c和.S文件
 $(1)_CFILES := $$(shell find $$($(1)_SRC_DIR) -name "*.c")
 $(1)_SFILES := $$(shell find $$($(1)_SRC_DIR) -name "*.S")

#.c对应的.o
 $(1)_COBJS := $$(patsubst $$($(1)_SRC_DIR)%.c,$$($(1)_OBJ_DIR)%.o,$$($(1)_CFILES))
#.S对应的.o
 $(1)_SOBJS := $$(patsubst $$($(1)_SRC_DIR)%.S,$$($(1)_OBJ_DIR)%.o,$$($(1)_SFILES))
#.o
 $(1)_OBJS := $$($(1)_SOBJS) $$($(1)_COBJS)

#nemu
 $(1)_BIN := $$($(1)_OBJ_DIR)/$(1)
#编译、汇编选项
 $(1)_CFLAGS = $(CFLAGS) -I$$($(1)_INC_DIR) $(2)
 $(1)_ASFLAGS = -m32 -MMD -c -I$$($(1)_INC_DIR) -I$(LIB_COMMON_DIR)
#从.c生成.o
 $$($(1)_OBJ_DIR)%.o: $$($(1)_SRC_DIR)%.c
     $$(call make_command, $(CC), $$($(1)_CFLAGS), cc $$<, $$<)
#从.S生成.o
 $$($(1)_OBJ_DIR)%.o: $$($(1)_SRC_DIR)%.S
     $$(call make_command, $(CC), $$($(1)_ASFLAGS), as $$<, $$<)

 -include $$($(1)_OBJS:.o=.d)
 endef
```

链接：

在`nemu/Makefile.part`里，我们看见了nemu相关的链接命令

```
 $(nemu_BIN): $(nemu_OBJS)
     $(call make_command, $(CC), $(nemu_LDFLAGS), ld $@, $^)
     $(call git_commit, "compile NEMU")
```

这会将所有更新好的.o文件链接起来