PA2 后半阶段实验报告

14307130078 张博洋

一、实验进度

完成了 PA2 所有内容。

二、部分选答题

1. 用 L 语言写出两整数相加的程序

程序思路很简单：每次做 x 自减、a 自增两个步骤，直到 x 为零；再对 y 做相同的事情即可。用 C 语言代码描述是这样：

```
while (x != 0) x--, a++;
while (y != 0) y--, a++;
```

把这段 C 语言程序翻译成 L 语言即可：

```
loop1:
JNZ x, loop2
DEC x
INC a
JNZ b, loop1 // b is always zero, jnz always jumps

loop2:
JNZ y, done
DEC y
INC a
JNZ b, loop2

done:
// a should be x+y here
```

2. 捕捉死循环

这是著名的停机问题，数学家已经从理论上证明了不可能存在这样的程序。但是，对于内存、寄存器有限的机器（即总状态数有限），从理论上是可以判定是否停机的。只要记住程序执行中的所有状态，一旦发现当前状态与历史某状态相同，则可判定停机。实际上，这种方法不可能实现出来。假设内存只有 1MB，状态数就高达 2^{8M}，这个状态数大约是 $10^{2525222}$，已经远超宇宙中所有原子的数量 10^{81} 了。

三、部分选答题

1. 编译与链接 1

（1）去掉 static

错误信息如下：

```
+ ld obj/nemu/nemu
obj/nemu/cpu/exec/data-mov/push.o：在函数‘instr_fetch’中：
/home/zby/src/fdu-ics/programming-assignment/nemu/include/cpu/helper.h:10: `instr_fetch'被多次定义
obj/nemu/monitor/cpu-exec.o:/home/zby/src/fdu-ics/programming-assignment/nemu/include/cpu/helper.h:10：第一次在此定义
obj/nemu/cpu/exec/data-mov/cbw_cwde.o：在函数‘instr_fetch’中：
/home/zby/src/fdu-ics/programming-assignment/nemu/include/cpu/helper.h:10: `instr_fetch'被多次定义
obj/nemu/monitor/cpu-exec.o:/home/zby/src/fdu-ics/programming-assignment/nemu/include/cpu/helper.h:10：第一次在此定义
```

可以看出是链接时发生了错误。这是因为每一个包含 helper.h 的.c 文件中都定义了一个名字叫做 instr_fetch 的函数实体，且没有 static 说明符。因此每一个这样的.c 对应的.o 文件中都会有强符号 instr_fetch 出现，链接

时多个强符号冲突导致出错。

（2）去掉 inline

错误信息如下：

```
+ cc nemu/src/cpu/exec/data-mov/push.c
In file included from nemu/include/cpu/exec/helper.h:4:0,
                 from nemu/src/cpu/exec/data-mov/push.c:1:
nemu/include/cpu/helper.h:10:17: error: ‘instr_fetch’ defin
ed but not used [-Werror=unused-function]
 static uint32_t instr_fetch(swaddr_t addr, size_t len) {
                 ^
cc1: all warnings being treated as errors
```

可以看出是编译时发生了错误。这是因为在某一些包含了 helper.h 的.c 文件中并没有用到 instr_fetch 这个函数。由于该函数带有 static 说明符，且不带 inline 说明符，编译时还开启了-Wall 选项，因此编译器会报一个“该函数没有被用到”的警告。但是由于开启了-Werror，该警告被强制变成错误，导致编译失败。

2．编译与链接 2

（1）在 common.h 中添加而生成的 dummy 实体数量

通过 readelf -s obj/nemu/nemu 可以看到最终生成的 nemu 的可执行文件的符号表，只要统计这张表中 dummy 出现的次数即可知道生成的 dummy 实体的数量。

```
$ readelf -s obj/nemu/nemu | grep ' dummy$' | wc -l
68
```

可以看出共有 68 个实体。

（2）在 debug.h 中添加而生成的 dummy 实体数量

通过相同的方法得到实体的数量仍然为 68。这是因为所有引用了 common.h 的.c 文件都引用了 debug.h，并且所有引用了 debug.h 的.c 文件都引用了 common.h 文件。因此在同一个.c 文件中，出现了两句 volatile static int dummy，在 C 语言规范中，这是一种“临时定义”，这样的“临时定义”允许出现多次，但是真正定义出来的实体只有一个，所以实体的数目并没有发生变化。

（3）初始化 dummy 导致的错误

错误信息如下：

```
+ cc nemu/src/monitor/cpu-exec.c
In file included from nemu/include/nemu.h:4:0,
                 from nemu/include/cpu/helper.h:4,
                 from nemu/src/monitor/cpu-exec.c:2:
nemu/include/common.h:75:21: error: redefinition of ‘dummy’
 volatile static int dummy = 0;
                     ^
In file included from nemu/include/common.h:13:0,
                 from nemu/include/nemu.h:4,
                 from nemu/include/cpu/helper.h:4,
                 from nemu/src/monitor/cpu-exec.c:2:
nemu/include/debug.h:40:21: note: previous definition of ‘du
mmy’ was here
 volatile static int dummy = 0;
```

可以看出这是因为出现两句 volatile static int dummy = 0 导致的

重定义错误。在 C 语言规范中，这是一种“定义”，在一个.c 文件中只能出现一次，因此是编译错误。

3. 了解 makefile

Makefile 的规则形式一般如下：

```
目标项：依赖项
    生成目标项的命令
```

通过阅读 Makefile 的内容可以发现，主 Makefile 引用了 config 目录下的 Makefile.build 文件。其中定义了名为 make_common_rules 的多行变量。在 nemu/Makefile.part 文件的第二行，利用 eval 函数和 call 函数，生成了对$(nemu_BIN)即 obj/nemu/nemu 的一系列生成规则。

仔细观察 make_common_rules 的内容可以发现，它首先通过调用 shell 命令执行 find -name "*.c"，定义了变量$(1)_CFILES。即在这个变量里存储了所有相关的.c 文件的文件名。接着又通过 patsubst 函数定义了$(1)_COBJS，它存储了$(1)_CFILES 变量中.c 对应的.o 的文件名。最后通过重写隐式规则，确定了由.c 生成.o 的命令。在 Makefile.build 的末尾处，还用-include $$($(1)_OBJS:.o=.d)解决了.c 与.h 文件之间的依赖关系。

由.c 生成.o 文件的规则是由 make_common_rules 生成的，而由.o 生成最终的可执行文件的规则，定义在 nemu/Makefile.part 中的第六行。可以看出，在依赖了所有.o 文件后，调用$(CC)命令进行了链接，生成了最终的可执行文件。