

PA2 实验报告

姓名：何伟 学号：171240537

2019 年 1 月 6 日

摘要

在 debian 虚拟机中完成了 PA2 的所有必做内容。

1 实验进度

1.1 PA2.1

PA2.1 实现 6 条指令成功运行 dummy。

1.2 PA2.2

PA2.2 实现更多的指令。在过第一个测试 add 的时候始终难以找到 bug，于是先去做了基础设施的 diff-test，在 diff-test 的帮助下实现了更多的指令并过了除 string 和 hello-str 之外的 cputest。之后实现了库函数，过了 string 和 hello-str。

1.3 PA2.3

PA2.3 发现了很多 2.2 的 bug 并修正，之后实现四个阶段的 IOE，成功在 nemu 中运行打字小游戏和马里奥。

2 必做题

2.1 编译与链接 static inline

选取了 rtl.h 中的较常用的 rtl_j 和 rtl_push 进行了测试，部分结果如下

```
110  
111 void interpret_rtl_j(vaddr_t target) {  
112     cpu.eip = target;  
113     decoding_set_jump(true);  
114 }
```

图 1: 图一

```
+ CC src/main.c
+ LD build/nemu
build/obj/cpu/exec/system.o: In function `interpret_rtl_j':
/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: multiple definition
of `interpret_rtl_j'
build/obj/cpu/exec/logic.o:/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: first defined here
build/obj/cpu/exec/exec.o: In function `interpret_rtl_j':
/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: multiple definition
of `interpret_rtl_j'
build/obj/cpu/exec/logic.o:/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: first defined here
build/obj/cpu/exec/special.o: In function `interpret_rtl_j':
/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: multiple definition
of `interpret_rtl_j'
build/obj/cpu/exec/logic.o:/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: first defined here
build/obj/cpu/exec/cc.o: In function `interpret_rtl_j':
/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: multiple definition
of `interpret_rtl_j'
build/obj/cpu/exec/logic.o:/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: first defined here
build/obj/cpu/exec/prefix.o: In function `interpret_rtl_j':
/home/hw/ics2018/nemu/./include/cpu/rtl.h:112: multiple definition
```

图 2: 图二

```
110
111 static void interpret_rtl_j(vaddr_t target) {
112     cpu.eip = target;
113     decoding_set_jump(true);
114 }
115
```

图 3: 图三

```
hw@debian:~/ics2018/nemu$ make run
+ CC src/cpu/exec/logic.c
In file included from ./include/cpu/decode.h:6:0,
                  from ./include/cpu/exec.h:9,
                  from src/cpu/exec/logic.c:1:
./include/cpu/rtl.h:111:13: error: `interpret_rtl_j' defined but not
used [-Werror=unused-function]
    static void interpret_rtl_j(vaddr_t target) {
                  ^~~~~~
cc1: all warnings being treated as errors
Makefile:31: recipe for target 'build/obj/cpu/exec/logic.o' failed
make: *** [build/obj/cpu/exec/logic.o] Error 1
hw@debian:~/ics2018/nemu$
```

图 4: 图四

从中我们可以看到去掉 inline，编译运行会报错，去掉 static 和 inline 也会报错，去掉 static 之后程序可以正常编译运行。

先看一下去掉 static 之后发生的错误，函数被定义了但未被使用，编译运行 nemu 加入了双 W 选项，警告变成了错误，这个错误很明显。但是加上了 inline 之后就可以正常编译运行。我们知道 inline 关键字是建议编译器做内联展开处理，所以推测加上了

inline 之后在编译环节并不会处理这个函数。同时去掉 static 和 inline 也报错，大概的报错信息就是有重复定义的产生。下面是进行了一些简单的验证。

```
1 #include <stdio.h>
2
3 static int a = 10;
4
5 int main()
6 {
7     int b=10;
8     printf("%d\n",b);
9     return 0;
10 }
```

图 5: 图五

```
hw@debian:~/code/pa$ gcc -Wall -Werror static.c
static.c:3:12: error: 'a' defined but not used [-Werror=unused-variable]
static int a = 10;
           ^
cc1: all warnings being treated as errors
hw@debian:~/code/pa$
```

图 6: 图六

```
1 #include <stdio.h>
2
3 static int fin(){return 1;};
4
5 int main()
6 {
7     int b=10;
8     printf("%d\n",b);
9     return 0;
10 }
```

图 7: 图七

```
hw@debian:~/code/pa$ gcc -Wall -Werror static.c
static.c:3:12: error: 'fin' defined but not used [-Werror=unused-function]
    static int fin(){return 1;};
               ^~~
cc1: all warnings being treated as errors
hw@debian:~/code/pa$
```

图 8: 图八

上面的例子中我们声明了一个静态变量 `a`，和一个静态函数，但是没有使用，在加上 `-Wall -Werror` 的 `gcc` 编译命令之后报错，上网查了之后得知 `-Wall` 选项会对函数中声明了但是未使用的函数，局部变量给出警告。

在庞大的 `nemu` 代码中，一旦源码中函数名称定义相同，就会出现编译出错。因此，需要引入一些封装的特性，限制源码中函数和变量作用的空间。在 `rtl` 中添加 `static` 关键字，其作用范围将缩小到仅仅为当前的文件，而不是整个系统。这样避免了函数定义相同的出错。

但是只是用 `static` 会出现声明了未使用的问题，于是加上了 `inline`。一方面，`inline` 建议编译器对函数进行内联展开处理。我们知道，函数调用需要时间和空间开销，调用函数实际上将程序执行流程转移到被调函数中，被调函数的代码执行完后，再返回到调用的地方。这种调用操作要求调用前保护好现场并记忆执行的地址，返回后恢复现场，并按原来保存的地址继续执行。对于较长的函数这种开销可以忽略不计，但对于一些函数体代码很短，又被频繁调用的函数，就不能忽视这种开销。引入内联函数正是为了解决这个问题，提高程序的运行效率。另一方面，解决了 `static` 函数声明未使用的警告。

```
1 #include <stdio.h>
2
3 int fun(){return 1;};
4
5 int main()
6 {
7     int b=10;
8     printf("%d\n",b);
9     return 0;
10 }
```

图 9: 图九

```
3  .globl  fun
4  .type   fun, @function
5  fun:
6  .LFB0:
7      .cfi_startproc
8      pushq   %rbp
9      .cfi_def_cfa_offset 16
10     .cfi_offset 6, -16
11     movq     %rsp, %rbp
12     .cfi_def_cfa_register 6
13     movl     $1, %eax
14     popq     %rbp
15     .cfi_def_cfa 7, 8
16     ret
17     .cfi_endproc
18 .LFE0:
19     .size    fun, .-fun
20     .section .rodata
21 .LC0:
22     .string  "%d\n"
23     .text
24     .globl  main
25     .type   main, @function
26 main:
27 .LFB1:
28     .cfi_startproc
```

图 10: 图十

```
File Edit View Search Terminal Help
1  #include <stdio.h>
2
3  inline int fun(){return 1;};
4
5  int main()
6  {
7      int b=10;
8      printf("%d\n",b);
9      return 0;
10 }
~
```

图 11: 图十一

```

1  .file "static.c"
2  .section .rodata
3  .LC0:
4  .string "%d\n"
5  .text
6  .globl main
7  .type main, @function
8  main:
9  .LFB1:
10 .cfi_startproc
11 pushq %rbp
12 .cfi_def_cfa_offset 16
13 .cfi_offset 6, -16
14 movq %rsp, %rbp
15 .cfi_def_cfa_register 6
16 subq $16, %rsp
17 movl $10, -4(%rbp)
18 movl -4(%rbp), %eax
19 movl %eax, %esi
20 leaq .LC0(%rip), %rdi
21 movl $0, %eax
22 call printf@PLT
23 movl $0, %eax
24 leave
25 .cfi_def_cfa 7, 8
26 ret

```

图 12: 图十二

在上面的例子中，首先声明了一个 int 型函数，用 gcc -S 选项生成汇编代码.s 文件，查看后发现其中对 fun 函数进行了编译，在加上了 inline 之后并没有处理，这与我们的预期是一致的，inline 消除了 static 函数未使用的警告错误。

关于同时删除 static 和 inline 的错误，在前面已经解释了不加 static 的函数是全局可见的，因此在编译成链接文件时，会被多次编译，从而导致了 multiple definition。于是我在一个.h 文件中写了一个函数 void fun()，在另外两个文件的函数中调用 fun()，最后在 main.c 调用这两个函数。

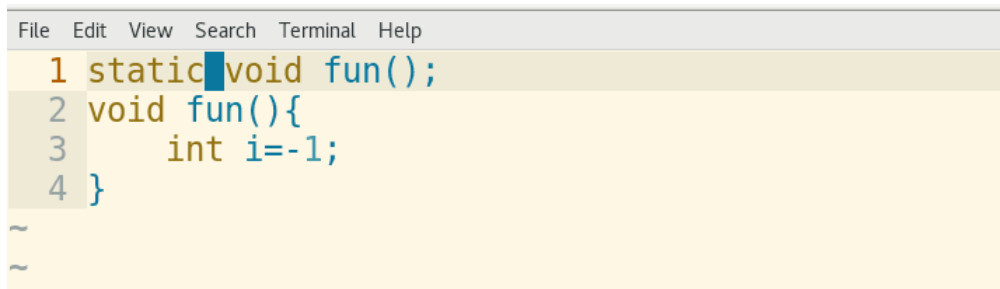
```

hw@debian:~/code/pa/static$ gcc main.c bar.c foo.c -o main
/tmp/cc9dom1L.o: In function `fun':
foo.c:(.text+0x0): multiple definition of `fun'
/tmp/ccxZuNV9.o:bar.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
hw@debian:~/code/pa/static$

```

图 13: 图十三

发生了同样的错误



```
File Edit View Search Terminal Help
1 static void fun();
2 void fun(){
3     int i=-1;
4 }
~
~
```

图 14: 图十四



```
hw@debian:~/code/pa/static$ vim fun.h
hw@debian:~/code/pa/static$ gcc main.c bar.c foo.c -o main
hw@debian:~/code/pa/static$
```

图 15: 图十五

之后在 `fun()` 声明时加上了 `static`，编译通过。

`nemu` 中有着庞大数量的代码，我们需要 `static` 保证函数不会被重复编译链接，解决函数同名冲突，也需要 `inline` 解决定义未使用问题，同时以空间为代价，提高时间效率。

2.2 编译与链接 2

1. 因为我们声明的是未初始化的全局变量，所以在 ELF 文件信息中存放在 `.bss` 节，`.bss` 节会记录数据所需空间大小。在未声明变量时通过 `readelf -S nemu` 查看 `.bss` 节记录的大小为 `b78`(只记录了后四位)，之后再 `common.h` 中加上了局部变量并重新查看，计算两者的差值为 108，除 4 之后得到个数为 32 个。
2. 在 `debug.h` 中加了之后使用同样的方法进行统计得到的个数还是 32 个。并不知道原因，于是将 `debug.h` 中声明的 `dummy` 换成了 `dummy1` 重新查看 `.bss` 存的大小，差值变成了两倍。推测 `volatile static int` 声明了两个 `dummy` 变量，变量被赋予了全局的属性，一个是定义，一个是引用，是同一个变量，因此个数并没有变化。
3. 进行初始化之后 `make` 出错。

```

hw@debian:~/ics2018/nemu$ make
+ CC src/misc/logo.c
+ CC src/memory/memory.c
In file included from ./include/common.h:33:0,
                  from ./include/nemu.h:4,
                  from src/memory/memory.c:1:
./include/debug.h:7:21: error: redefinition of 'dummy'
volatile static int dummy=0;
                  ^~~~~
In file included from ./include/nemu.h:4:0,
                  from src/memory/memory.c:1:
./include/common.h:4:21: note: previous definition of 'dummy' was here
volatile static int dummy=0;
                  ^
Makefile:31: recipe for target 'build/obj/memory/memory.o' failed
make: *** [build/obj/memory/memory.o] Error 1

```

图 16: 图十六

看样子时定义冲突，但是加上了 `static` 是本地符号，也没有强弱之分，去掉 `volatile` 之后报错信息就变了。`volatile` tells the compiler that your variable may be changed by other means. 在网上看到了这样一句话，`volatile` 大概就是将变量共享并使得它不被优化。这里用了 `volatile` 关键字之后有初始化，便有了强符号的属性，在多次编译链接时便会报错。

2.3 Makefile

make: "GNU make utility to maintain groups of programs"

Makefile: "You need a file called a *Makefile* to tell make what to do. Most often, the Makefile tells make how to compile and link a program."

这是手册里看到的两句话，通俗地说，Makefile 是一种脚本，用于指导 make 的编译与链接。基本的 Makefile 文件包含三个部分，目标文件，生成目标文件所需的依赖文件和编译命令。

当我们在 nemu 中 make 的时候，首先会找到名为 Makefile 的文件，随后才是执行 Makefile 文件，分为 Makefile 工作部分和编译和链接的部分。在 Makefile 工作的部分，Makefile 首先会根据 "include" 指令把读取工作目录中的文件，处理内建的变量、明确规则和隐含规则，并建立所有目标和依赖之间的依赖关系结构链表。随后 make 会执行 Makefile 中的终极目标，根据之前的依赖关系进行一系列的编译工作。所有的编译和链接都会围绕最终目标展开，根据最终文件的要求，依赖，进行编译与链接，最终生成目标文件。nemu 的 Makefile 中用到了 "include"，将 Makefile.git 贴进去，所以每次编译运行都会自动进行 git 的有关工作。一般目标文件都会写在靠前的位置。

```

29 # Compilation patterns
30 $(OBJ_DIR)/%.o: src/%.c
31     @echo + CC $<
32     @mkdir -p $(dir $@)
33     @$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
34

```

图 17: 图十七

上图中可以看出，终极目标是生成一些.o 文件，依赖于一些.c 文件，下面的是编译命令。后面还有一些 `make gdb`, `make run` 等其他的 `make` 命令。

Makefile 中用到了大量的变量，一方面，使脚本更加简洁，避免了很多长而繁琐的命令。其次，用到诸如系统变量“CFLAGS”这种的变量可以控制编译时的编译器参数，修改一些默认的隐藏规则。

3 实验心得

本次实验将 nemu 中的实现与 am 中运行环境联系起来，环环相扣。通过自己实现指令，仔细阅读 i386 手册，对计算机的架构有了更加深入地了解。实验过程中也遇到了很多致命地 bug，让人头秃。在开始进行 2.1 实验的时候，无从下手，对计算机中命令的执行方式一窍不通，看了快两天的讲义，手册，源码才开始动手，开头非常难。指令的具体实现也还行。但是也走了不少歪路，一开始不理解指令宽度的含义，凭感觉瞎写，后面稍微懂了一些。还有一些是关于符号位扩展和 EFLAGS 寄存器的，可能也是出现 bug 的重灾区了，也许当前的 test 过了，也没有出现 bug，但实现的可能就是不对，隐藏在代码之中。另外，可能理解错了讲义的意思，函数的封装与调用，在 rtl 函数中调用了很多其他的 rtl 函数，调用经常会用到寄存器，寄存器多了，在写执行函数时就可能会产生冲突，在这方面吃了不少亏。

总之，PA2 的实验让我们更加深入的了解计算机的工作原理和工作步骤。其次，再次认识基础设施的重要性，很难想象没有 diff-test 的 PA2.2，太可怕！