



南開大學
Nankai University

计算机学院
计算机系统设计实验报告

PA1: 开天辟地的篇章：最简单的计算机

姓名：李颖
学号：2110939
专业：计算机科学与技术

2024 年 3 月 18 日

目录

1 实验概述	3
1.1 实验目的	3
1.2 实验内容	3
2 阶段一：NEMU 与简易调试器	3
2.1 理解 NEMU 框架	3
2.1.1 NEMU 框架与执行过程	3
2.1.2 问题 1: 实现正确的寄存器体	4
2.1.3 问题 2: CPU 究竟要执行多久	5
2.1.4 问题 3: 谁来指示程序的结束	5
2.2 实现简易调试器	6
2.2.1 解析命令	6
2.2.2 功能 1: 单步执行	6
2.2.3 功能 2: 打印寄存器	7
2.2.4 功能 3: 扫描内存	8
3 阶段二：表达式求值	9
3.1 词法分析	10
3.1.1 token 类型定义	10
3.1.2 优先级定义及 token 识别	11
3.2 递归求值	12
3.2.1 判断表达式是否被括号包围	12
3.2.2 寻找主导操作符 dominant operator	13
3.2.3 递归计算求值	14
3.2.4 测试结果	16
3.3 带有负数的递归求值	16
3.4 更复杂的表达式求值	17
4 阶段三	19
4.1 实现监视点池的管理	19
4.2 问题 1: static 的含义及作用	20
4.3 实现监视点	20
4.3.1 打印监视点信息	20
4.3.2 设置监视点	20
4.3.3 删除监视点	21
4.3.4 实现暂停操作	22
4.4 测试结果	23
4.5 问题 2: 一点也不能长?	23
4.6 问题 3: 随心所欲”的断点	23
4.7 问题 4:NEMU 的前世今生	24

5 必答题	24
5.1 问题 1: 查阅 i386 手册, 标明阅读范围	24
5.2 问题 2: shell 命令统计代码行数	25
5.3 问题 3: gcc 编译选项-Wall 与-Werror 的作用	25

1 实验概述

1.1 实验目的

PA 的目的是要实现 NEMU, 一款经过简化的 x86 全系统模拟器。NEMU 就是一个虚拟出来的计算机系统, 物理计算机中的基本功能, 在 NEMU 中都是通过程序来实现的。利用 NEMU 模拟出来的 x86 环境, 我们可以在其中执行其他程序。

1.2 实验内容

在 NEMU 中实现一个最简单的计算机 TRM, 理解并掌握 NEMU 框架代码, 理解 NEMU 的执行过程, 并完善相关代码。

在 monitor 中实现一个给 NEMU 使用的简易的调试器, 为后续实验做铺垫。相关的指令格式和内容如下所示。

命令	格式	使用举例	说明
帮助 (1)	help	help	打印命令的帮助信息
继续运行 (1)	c	c	继续运行被暂停的程序
退出 (1)	q	q	退出 NEMU
单步执行	si [N]	si 10	让程序单步执行 N 条指令后暂停执行, 当 N 没有给出时, 缺省为 1
打印程序状态	info SUBCMD	info r info w	打印寄存器状态 打印监视点信息
表达式求值	p EXPR	p \$eax + 1	求出表达式 EXPR 的值, EXPR 支持的运算请见调试中的表达式求值小节
扫描内存 (2)	x N EXPR	x 10 \$esp	求出表达式 EXPR 的值, 将结果作为起始内存地址, 以十六进制形式输出连续的 N 个 4 字节
设置监视点	w EXPR	w *0x2000	当表达式 EXPR 的值发生变化时, 暂停程序执行
删除监视点	d N	d 2	删除序号为 N 的监视点

2 阶段一：NEMU 与简易调试器

2.1 理解 NEMU 框架

2.1.1 NEMU 框架与执行过程

NEMU 框架如下所示：

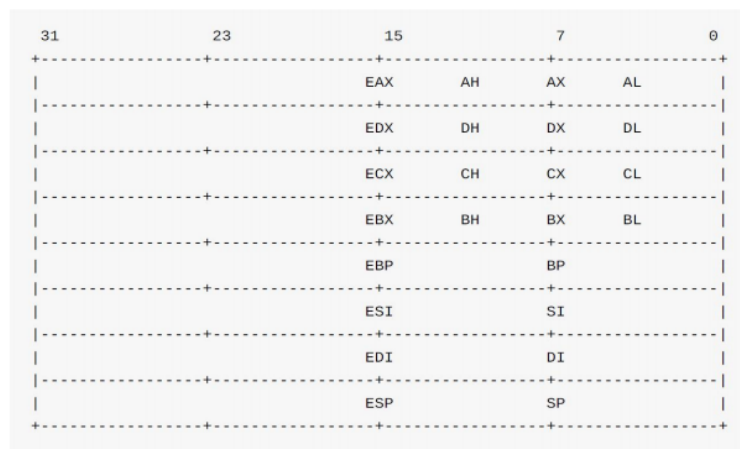
1. CPU：中央处理器，是负责处理数据的核心电路单元，负责运算；CPU 中有寄存器，将正在处理的数据短暂地放置其中；
2. memory：用于放置程序的容器，将指令置于存储器中，就可以让计算机执行一系列指令序列。
3. monitor：负责与 GNU/Linux 进行交互，带有调试器功能，为 NEMU 的调试提供了方便的途径。

NEMU 执行过程如下所示：

1. NEMU 开始执行时，先调用 `init_monitor()` 函数，进行一些和 `monitor` 相关的初始化工作；
2. 通过调用 `load_img()` 函数读入带有客户程序的镜像文件，让 `monitor` 直接把程序镜像 `guest prog` 读入到一个固定的内存位置 `0x100000`；
3. 调用 `restart()` 函数模拟“计算机启动”的功能，记性一些和“计算机启动”相关的初始化工作；
4. 将 `$eip` 的初值设置为刚才约定的内存位置 `0x100000`，让 CPU 从约定的内存位置开始执行程序。

2.1.2 问题 1：实现正确的寄存器体

由于寄存器结构体 `CPU_state` 没有被正确实现，因此 `make run` 时会报错。



x86 的通用寄存器结构如上图所示。由图可知，32 位的 EAX 寄存器也可以分为 16 位的 AX 寄存器或两个 8 位的 AH 和 AL 寄存器。因此，在 `cpu/reg.h` 中，需要使用 `union` 将这 8 个通用寄存器的 32 位、16 位、两个 8 位寄存器组织起来，相关代码如下所示。

```

1  union{
2  union {
3      uint32_t _32;
4      uint16_t _16;
5      uint8_t _8[2];
6  } gpr[8];
7
8  struct{
9      rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
10 };
11 };

```

实现了正确的寄存器体后，执行 `make run` 命令，显示 “Welcome to NEMU!”，成功进入 NEMU 内部，如图所示。

```

ly@ubuntu:~/Desktop/ics2017/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default build-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:35:05, Mar  7 2024
For help, type "help"

```

2.1.3 问题 2：CPU 究竟要执行多久

在 `cmd_c()` 函数中，调用 `cpu_exec()` 的时候传入了参数-1，原因及运行时间如下所示。

查看函数对应的代码可知，`cpu_exec()` 函数接受一个 `uint64_t` 类型的参数 `n`，用于指定需要执行的指令数量。当传入参数-1 时，由于 `n` 无符号，当于传入一个非常大的正整数，导致循环执行大量指令。函数会进入一个循环，在循环中，每执行一条指令后会检查当前的状态，包括是否需要打印指令、是否需要更新设备状态等。执行指令直到满足某种条件，如果在执行过程中状态发生变化（如遇到异常、中断等），则会提前退出循环。

综上所述，NEMU 的执行次数与传入参数有关。而传入参数后，NEMU 将不断执行指令，直到遇到以下情况之一，才会退出指令执行的循环：

1. 达到要求的循环次数；
2. 客户程序执行了 `nemu_trap` 指令，NEMU 会输出：“nemu: HIT GOOD TRAP at eip = 0x00100026”

2.1.4 问题 3：谁来指示程序的结束

程序执行到 `main()` 函数的返回处并不是程序结束的唯一标志。在大多数情况下，程序执行完 `main()` 函数后会自动结束，因为操作系统会在 `main()` 函数返回后将程序的控制权交还给操作系统，从而导致程序结束。

但实际上，程序的结束可以通过多种方式来指示。例如，在 C 语言中，程序可以通过调用 `exit()` 函数来显式地指示程序的结束，同时传递一个退出码。除此之外，程序也可以通过发生致命错误、接收到信号、调用 `abort()` 函数等方式来终止程序的执行。

例如，以下的 c 语言程序就并非在 `main()` 函数中结束，而是在 `fun()` 函数中结束。

```

1 #include "stdlib.h"
2 void fun ( void ) ;
3 int main ( void ){
4     fun();
5     return 0;
6 }
7 void fun ( void ){
8     exit(1);
9 }

```

此外，通过使用 `atexit` 函数，可以用来注册在程序即将退出时执行的清理函数，这样可以确保在程序结束之前执行一些必要的操作，比如关闭文件、释放内存空间等。`atexit` 函数是标准 C 新增的函数，它“注册”一个函数，使这个函数将在 `exit` 函数被调用时或者当 `main` 函数返回时被调用。当程序异常终止时（如调用 `abort` 或 `raise`），通过它注册的函数不会被调用。编译器必须至少至少允许程序员注册 32 个函数。如果注册成功，`atexit` 返回 0，否则返回非 0，没有办法取消一个函数的注册。在

exit 所执行的任何标准清理操作之前。被注册的函数按照与注册顺序相反的顺序被依次调用。通过注册这些清理函数，程序可以在退出时自动执行这些操作，避免遗漏或忘记执行清理步骤。

2.2 实现简易调试器

简易调试器是 NEMU 中一项非常重要的基础设施。由于 NEMU 是一个用来执行其它客户程序的程序，因此它可以随时了解客户程序执行的所有信息。由于直接从 GDB 中获取信息比较困难，因此需要在 NEMU 中实现简易调试器来获取进程的各种信息，用以知道程序是否出现问题。

本次实验需要在 monitor 中实现一个具有多种功能的简易调试器，在当前阶段，只需实现单步执行、打印寄存器、扫描内存三种功能。

2.2.1 解析命令

在使用调试器时，我们需要在 NEMU 中输入指定命令，NEMU 解析命令并做出响应。因此，解析命令是实现简易调试器的最基本工作。NEMU 通过 readline 库与用户交互，使用 readline() 函数从键盘上读入命令。与 gets() 相比，readline() 提供了“行编辑”的功能，最常用的功能就是通过上、下方向键翻阅历史记录。事实上，shell 程序就是通过 readline() 读入命令的。readline 的功能如下所示。

1. 提供了命令行编辑功能，包括光标移动、删除、插入等操作；
2. 可以使用上下方向键来查看历史输入记录；
3. 支持自动补全功能，根据用户输入的前缀自动完成命令或文件名；
4. 允许用户定义自定义快捷键和命令别名。

在 NEMU 的框架代码中，我们引入了 readline 库来读入用户命令，并将每一行的开头设置为“(NEMU)”。

从键盘上读入命令后，NEMU 需要解析该命令，然后执行相关的操作。解析命令的工作是通过一系列的字符串处理函数来完成的，在本次实验中，主要用到了如下两个函数来解析命令。

1. strtok 函数，用于将输入的字符串按照指定的分隔符进行分割，返回分割后的第一个标记。使用方法如下所示：

- 在第一次调用 strtok() 时，需传入待分割的字符串和分隔符字符串。例如：strtok(str, " ")。
- 后续调用 strtok(NULL, delimiters) 来获取下一个标记，其中 delimiters 为分隔符字符串。
- 最后一次调用 strtok(NULL, delimiters) 时，将返回 NULL，表示没有更多标记

2. sscanf 函数，从一个字符串中按照指定格式提取数据，并将提取的数据存储到指定的变量中。

sscanf() 函数的原型为 int sscanf(const char *str, const char *format, ...)。第一个参数是待解析的字符串，第二个参数是格式化字符串，后面的参数是要存储数据的变量。与 scanf() 函数类似，但是 sscanf() 不是从标准输入中读取数据，而是从指定字符串中读取数据。

2.2.2 功能 1：单步执行

命令	格式	使用举例	说明
单步执行	si [N]	si 10	让程序单步执行 N 条指令后暂停执行，当 N 没有给出时，缺省为 1

单步执行的指令格式为 `si N`，若未输入子命令 `N`，则只执行一步，否则执行 `N` 步。由于 `c` 命令为继续运行，查看 `cmd_c` 函数，可知该函数中调用了 `cpu_exec(-1)` 函数。在之前已经回答过，传入的参数即为执行步数，传入 `-1` 意味着继续运行直至程序结束或遇到异常。因此，在单步执行 `cmd_si` 中，也可以利用 `cpu_exec()` 函数来执行指定步数。

相应的代码如下所示，首先在 `cmd_table` 中补充命令介绍和对应函数，再在具体函数中使用 `strtok` 解析命令，最后调用 `cpu_exec()` 执行指定步数。

```

1 cmd_table [] = {{ "si", "Execute the N instructions step by step", cmd_si }, };
2
3 static int cmd_si(char *args){
4     char *arg = strtok(NULL, " ");
5
6     if(arg!=NULL)
7         cpu_exec(atoi(arg)); //执行N步
8     else
9         cpu_exec(1); //单步执行
10    return 0;
11 }

```

成功实现后，在 NEMU 中的使用示例如图所示。

```

(nemu) q
ly@ubuntu:~/Desktop/ics2017/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default b
uild-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:35:05, Mar  7 2024
For help, type "help"
(nemu) si 5
100000: b8 34 12 00 00      movl $0x1234,%eax
100005: b9 27 00 10 00      movl $0x100027,%ecx
10000a: 89 01              movl %eax,(%ecx)
10000c: 66 c7 41 04 01 00   movw $0x1,0x4(%ecx)
100012: bb 02 00 00 00      movl $0x2,%ebx
(nemu) si
100017: 66 c7 84 99 00 e0 ff ff 01 00   movw $0x1,-0x2000(%ecx,%ebx,4)
(nemu) si 10
nemu: HIT GOOD TRAP at eip = 0x00100026
(nemu)

```

2.2.3 功能 2：打印寄存器

命令	格式	使用举例	说明
打印程序状态	info SUBCMD	info r	打印寄存器状态

打印程序状态 “info SUBCMD” 这个命令有 “r” 和 “w” 两种子命令，在阶段一我们只实现 “info r” 打印寄存器状态命令。只需要先用 `strcmp()` 函数比较子命令是否为 “r”，然后使用 `printf()` 函数打印出八个通用寄存器的名称和内容即可。

打印寄存器状态涉及的函数和定义如下所示。

- 寄存器名称：static inline const char *reg_name(int index, int width)
- 寄存器内容：#define reg_l(index) (cpu.gpr[check_reg_index(index)]._32)

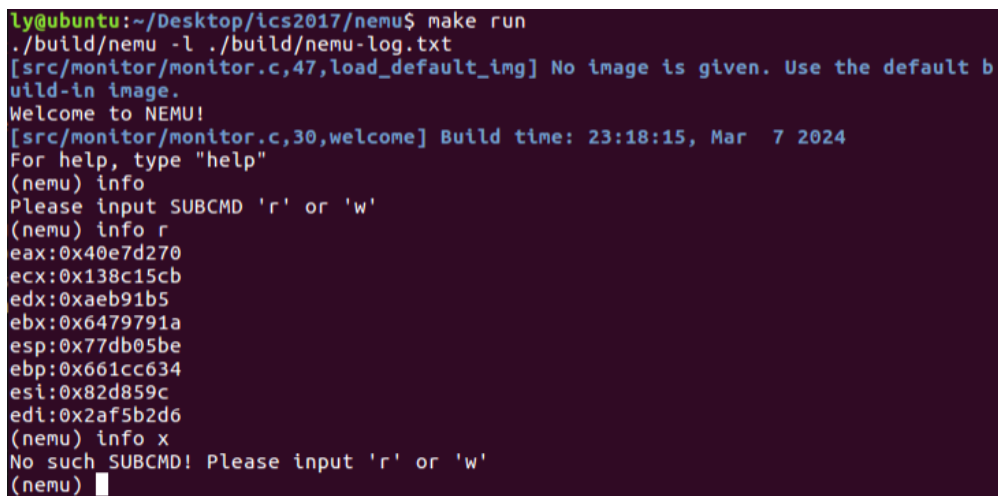
打印程序状态命令的补充介绍和具体函数实现代码如下所示。

```

1 cmd_table [] = {{"info", "Print the statement of programs, 'r' is register and 'w' is
  watchpoint", cmd_info}, };
2
3 static int cmd_info(char *args){
4     char *arg = strtok(NULL, "");
5     if(arg==NULL){
6         printf("Please input SUBCMD 'r' or 'w'\n");
7         return 0;
8     }
9
10    if(strcmp(arg, "r")==0)
11        for(int i=0; i<8; i++)
12            printf("%s:0x%x\n", reg_name(i, 4), reg_l(i));
13    else if(strcmp(arg, "w")==0)
14        ;
15    else
16        printf("No such SUBCMD! Please input 'r' or 'w'\n");
17    return 0;
18 }

```

成功实现后, 在 NEMU 中的使用示例如图所示。



```

ly@ubuntu:~/Desktop/ics2017/nemu$ make run
./build/nemu -l ./build/nemu-log.txt
[src/monitor/monitor.c,47,load_default_img] No image is given. Use the default build-in image.
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 23:18:15, Mar  7 2024
For help, type "help"
(nemu) info
Please input SUBCMD 'r' or 'w'
(nemu) info r
eax:0x40e7d270
ecx:0x138c15cb
edx:0xae91b5
ebx:0x6479791a
esp:0x77db05be
ebp:0x661cc634
esi:0x82d859c
edi:0x2af5b2d6
(nemu) info x
No such SUBCMD! Please input 'r' or 'w'
(nemu)

```

2.2.4 功能 3: 扫描内存

命令	格式	使用举例	说明
扫描内存	x N EXPR	x 10 \$esp	求出表达式 EXPR 的值, 将结果作为起始内存地址, 以十六进制

扫描内存命令格式为 “X N EXPR”, 即输出 EXPR 所表示的地址的 N 个连续的 4 字节。因此, 最需要解决的问题为 EXPR 所表示的值, 该部分在阶段二表达式求值中实现, 在本阶段, 只需要识别出形如 “0x100000” 的十六进制地址, 并输出该地址的内容即可。

使用 sscanf 函数对命令进行解析。解析出 “0x” 开头的字符串, 可得十六进制地址, 使用 atoi() 函数把字符串转换为整型。再使用 vaddr_read() 函数, 读出改地址上的 N 个 4 字节内容。打印程序

状态命令的补充介绍和具体函数实现代码如下所示。

```

1 cmd_table [] = {{ "x", "scan the memory, regard the result of expr as address and
   print", cmd_x}, };
2
3 static int cmd_x(char *args){
4     char *arg1 = strtok(NULL, " ");
5     if(arg1==NULL){
6         printf("Please input subcmp N\n");
7         return 0;
8     }
9     int i_arg1 = atoi(arg1);
10    char *arg2 = strtok(NULL, " ");
11
12    if(arg2==NULL){
13        printf("Please input subcmp expr\n");
14        return 0;
15    }
16    //uint32_t addr_begin = strtoul(arg2, NULL, 16);
17    uint32_t addr_begin ;
18    sscanf(arg2, "0x%x", &addr_begin);
19    printf("the result of expr is:%x\n", addr_begin);
20    for(int i=0; i<i_arg1; i++){
21        printf("0x%x ", vaddr_read(addr_begin, 1));
22        addr_begin+=1;
23    }
24    printf("\n");
25    return 0;
26 }

```

成功实现后，在 NEMU 中的使用示例如图所示。输入 `x 5 0x100000`，得到的结果如下所示。输入 `si` 进行单步执行，执行 `0x100000` 处的指令，可看到该地址的指令内容与 `x` 扫描内存时输出的内容一致，说明该功能可以正确实现。

```

(nemu) x 5 0x100000
the result of expr is:100000
0xb8 0x34 0x12 0x0 0x0
(nemu) si
100000: b8 34 12 00 00          movl $0x1234,%eax
(nemu) x 5 0x100005
the result of expr is:100005
0xb9 0x27 0x0 0x10 0x0
(nemu) si
100005: b9 27 00 10 00          movl $0x100027,%ecx
(nemu)

```

3 阶段二：表达式求值

命令	格式	使用举例	说明
表达式求值	p EXPR	p \$eax + 1	求出表达式 EXPR 的值

本阶段表达式求值的主要步骤为：先识别出表达式中的单元，并确定好运算符的优先级，再根据表达式的归纳定义进行递归求值。

3.1 词法分析

3.1.1 token 类型定义

利用正则表达式匹配识别出 token，首先需要定义 token 类型。本次需要实现的 token 类型如下所示。

- 寄存器名称
- 十进制整数、十六进制整数
- + - * /
- == !=
- && ||
- & | !
- (,)
- 空格串

因此，对应的 token 代码如下所示。

```

1 static struct rule {
2     char *regex;
3     int token_type;
4 } rules[] = {
5     {" +", TK_NOTYPE},    // spaces
6     {"\\+", '+'},        // plus
7     {"\\-", '-'},        // minus
8     {"\\*", '*'},        // multi/getval
9     {"\\/", '/'},        // div
10    {"==", TK_EQ},        // equal
11    {"!=", TK_NEQ},       // not equal
12    {"!", TK_NOT},        // log-not
13    {"\\|\\|", TK_LOR},   // log-or
14    {"&&", TK_LAND},       // log-and
15    {"\\|", TK_OR},       // calc-or
16    {"&", TK_AND},        // calc-and
17
18    {"\\$[eE][0-9a-zA-Z]{2}", REG}, // registers
19    {"[0-9]|([1-9][0-9]*)", DEC}, // decimal
20    {"0[xX][a-fA-F0-9]+", HEX}, // hex
21
22    {"\\(", '('},         // l-paren
23    {"\\)", ')'},         // r-paten
24 };

```

在编写规则时，需要注意规则的顺序。正则表达式会先匹配先定义的规则，再匹配后定义的，因此，当某些 token 的前缀一致时，规则的先后顺序会影响单词识别结果，例如“!=”有可能会被识别成“!”和“=”，导致后续递归求值时出现错误。需要注意顺序的 token 定义如下所示：

1. “!=”要在“!”之前定义；
2. REG 在整数之前定义，而对于整数来说，十六进制整数要在十进制整数之前定义，因此 HEX 有前缀“0x”；
3. “&&”和“||”要在“&”和“|”之前定义。

3.1.2 优先级定义及 token 识别

在下一步的递归求值中，需要利用运算符的优先级来确认计算顺序，因此，在识别 token 的时候，可以把每个运算符的优先级记录到 tokens 数组中。

首先，需要改变 token 结构体的定义，添加 int preference 变量，代表优先级。

```
1 typedef struct token {
2     int type;
3     char str[32];
4     int preference;
5 } Token;
```

在 token 结构体之前，增加 enum 类型，按优先级递增的顺序定义操作符优先级类型，如下所示。可以发现当 preference>1 时，该 token 才代表操作符，否则为括号或数值。每个优先级类型所代表的运算由以下的注释给出。

```
1 enum{
2     OP_VAL, //数字或寄存器
3     OP_PARN, //括号
4     OP_UADD, //+1 -1 单目运算
5     OP_MUL, //乘法除法
6     OP_ADD, //双目加法减法
7     OP_NOT, //!
8     OP_EQ, //== !=
9     OP_AND, //&
10    OP_OR, //|
11    OP_LAND, //&&
12    OP_LOR, //||
13};
```

定义好优先级类型后，在 make_token() 函数中，把输入的表达式识别成一个个 token，并标注优先级。switch 部分的代码如下所示。

```
1 switch (rules[i].token_type) {
2     case TK_NOTYPE: break;
3     case '+':
4     case '-':{
5         tokens[nr_token].type=rules[i].token_type;
6         tokens[nr_token].preference=OP_ADD;
```

```

7         ++nr_token;
8         break;
9     }
10    // ... 此处省略类似结构代码
11    }default: {
12        tokens[nr_token].type = rules[i].token_type;
13        nr_token++;
14    }
15    }
16    break;
17    }

```

由于指针的 token 结构为 *id，而 * 会在 case 中被识别为乘号，因此，在 token 全部识别完毕时，需要对指针进行额外的判断：若当前 token 为 “*”，并且前面无 token 或前面 token 类型为操作符，则说明该 token 的类型是指针。相关代码如下所示。

```

1    for(int i=0;i<nr_token;i++){
2        if(tokens[i].type=='*'&&((tokens[i-1].preference>1)||i==0)){
3            tokens[i].type = TK_POINT;
4        }
5    }

```

3.2 递归求值

根据 BNF 的定义，我们可以得到一种解决思路：先对短表达式求值，然后再对长表达式求值。因此，我们需要确认表达式中的主导操作符，并递归计算主导操作符左右两边的值 val1 和 val2，最后用这两个值进行主导操作符代表的运算。

可以将递归运算的函数定义为 static uint32_t eval(int l, int r, bool *success)，记录表达式左右两边的边界下标。假设找到的主导操作符下标为 op_index，则 val1=eval(l,op_index-1,success)，而 val2=eval(op_index+1,r,success)。直到递归的短表达式为寄存器或数字时，递归停止，返回对应的数值。

此外，如果需要计算的表达式（或短表达式）被一对括号包围，则需要对括号内的部分进行计算，递归为 eval(l+1,r-1,success)。因此，我们还需要编写函数来判断表达式是否被括号包围。

3.2.1 判断表达式是否被括号包围

判断表达式是否被括号包围，只需要看第一个符号和最后一个符号是否是左右括号即可。此外，还需要判断括号范围是否有效，需记录左右括号数量是否相等。

相关代码如下所示：使用 math 记录括号是否匹配，遍历所有 token 后，若 math 为 0 则说明括号匹配正确。最后判断头尾 token 即可。

```

1    bool check_parentheses(int l, int r, bool *success){
2        int match = 0;
3        for(int i=l;i<=r;i++){
4            if(tokens[i].type=='('){
5                match+=1;
6            }

```

```

7     else if(tokens[i].type==')'){
8         match--;
9     }
10    if(match<0){
11        //flag = false;
12        *success=false;
13        return false;
14    }
15 }
16
17 if(match==0){
18     *success=true;
19     if(tokens[l].type=='('&&tokens[r].type==')')
20         return true;
21 }
22 return false;
23 }

```

3.2.2 寻找主导操作符 dominant operator

在词法分析时，已经按照优先级从低到高的顺序，用 enum 定义了运算符优先级，并在识别 token 时记录到 preference 中。因此，只需要遍历 token，找到 preference 最大的运算符的下标即可。

相关代码如下所示。由于定义优先级时，0 和 1 代表数值和括号，因此只有 preference>1 才代表运算符。

```

1 static int find_dominant_operator(int l, int r, bool *success){
2     int max_level=-1, index=-1;
3     int match_paren=0;
4     for(int i=l; i<=r; i++){
5         if(tokens[i].type=='('){
6             match_paren++;
7             continue;
8         }
9         else if(tokens[i].type==')'){
10            match_paren--;
11            continue;
12        }
13
14        if(match_paren!=0)
15            continue;
16        int cur_level=tokens[i].preference;
17        if(cur_level<2) //not operator
18            continue;
19
20        if(cur_level>max_level){
21            max_level=cur_level;
22            index=i;
23        }

```

```

24     }
25     return index;
26 }

```

3.2.3 递归计算求值

找到主导操作符的下标后，记为 `op_index`，则 `val1=eval(l,op_index-1,success)`，而 `val2=eval(op_index+1,r,success)`。直到递归的短表达式为寄存器或数字时，递归停止，返回对应的数值。

递归计算时，需要用左右边界进行条件判断。若左边界 $>$ 右边界，说明表达式有误；若左边界 $=$ 右边界，说明表达式中只有一个 token，返回对应数值即可；若左边界 $<$ 右边界，则递归计算 `op` 左右两边的数值，再进行相应计算。

整个递归计算的代码如下所示。

```

1 static uint32_t eval(int l, int r, bool *success){
2     if(l>r){
3         *success=false;
4         return 0;
5     }
6     if(l==r){
7         *success=true;
8         switch(tokens[l].type){
9             case DEC:{
10                 int dec = atoi(tokens[l].str);
11                 return dec;
12             }
13             case HEX:{
14                 uint32_t hex;
15                 sscanf(tokens[l].str, "0x%x", &hex);
16                 return hex;
17             }
18             case REG:{
19                 if(strcmp(&tokens[l].str[l], "eip")==0)
20                     return cpu.eip;
21                 for(int i=0;i<8;i++){
22                     if(strcmp(&tokens[l].str[l], reg_name(i,4))==0)
23                         return reg_l(i);
24                 }
25                 printf("register's name has error\n");
26             }
27             default:{
28                 *success=false;
29                 return 0;
30             }
31         }
32     }
33     else if(check_parentheses(l, r, success))
34         return eval(l+1, r-1, success);
35     else{

```

```

36  int op_index=find_dominant_operator(l,r,success);
37  //printf("dominant operator's index:%d\n",op_index);
38  uint32_t val2=0;
39  val2=eval(op_index+1,r,success);
40  //printf("val2=%d\n",val2);
41  if(tokens[op_index].type==TK_NOT){
42      if(*success)
43          return !val2;
44      else
45          return 0;
46  }
47  else if(tokens[op_index].type==TK_POINT)
48      return vaddr_read(val2,4);
49
50  uint32_t val1=0;
51      if(l<=op_index-1)
52          val1=eval(l,op_index-1,success);
53  else
54      if(tokens[op_index].type!='+'&&tokens[op_index].type!='-')
55          val1=eval(l,op_index-1,success);
56  //printf("val1=%d\n",val1);
57  if(!*success)
58      return 0;
59
60  switch(tokens[op_index].type){
61      case '+':
62          return val1+val2;
63      case '-':
64          return val1-val2;
65      case '*':
66          return val1*val2;
67      case '/':{
68          if(val2==0){
69              printf("0 cannot be divided\n");
70              *success = false;
71              return 0;
72          }
73          else
74              return val1/val2;
75      }
76      case TK_EQ:
77          return val1==val2;
78      case TK_NEQ:
79          return val1!=val2;
80      case TK_LAND:
81          return val1&&val2;
82      case TK_LOR:
83          return val1||val2;
84      case TK_AND:

```



```

85     return val1&val2;
86 case TK_OR:
87     return val1|val2;
88 default:{
89     *success=false;
90     printf("Fail!\n");
91     return 0;
92 }
93 }
94 }
95 }

```

3.2.4 测试结果

在 NEMU 中的执行结果如下所示，可知普通四则运算、指针取地址中的值、取寄存器中的值都正确。

```

(nemu) p 1+3*(2--4)
[src/monitor/debug/expr.c,116,make_token] match rules[14] = "[0-9]|([1-9][0-9]*)"
" at position 0 with len 1: 1
[src/monitor/debug/expr.c,116,make_token] match rules[1] = "+" at position 1 with
len 1: +
[src/monitor/debug/expr.c,116,make_token] match rules[14] = "[0-9]|([1-9][0-9]*)"
" at position 2 with len 1: 3
[src/monitor/debug/expr.c,116,make_token] match rules[3] = "*" at position 3 with
len 1: *
[src/monitor/debug/expr.c,116,make_token] match rules[15] = "(" at position 4 with
len 1: (
[src/monitor/debug/expr.c,116,make_token] match rules[14] = "[0-9]|([1-9][0-9]*)"
" at position 5 with len 1: 2
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 6 with
len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 7 with
len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[14] = "[0-9]|([1-9][0-9]*)"
" at position 8 with len 1: 4
[src/monitor/debug/expr.c,116,make_token] match rules[16] = ")" at position 9 with
len 1: )
the result is: 19

```

```

(nemu) p *0x100000
[src/monitor/debug/expr.c,116,make_token] match rules[3] = "*" at position 0 with
len 1: *
[src/monitor/debug/expr.c,116,make_token] match rules[13] = "0[xX][a-fA-F0-9]+"
at position 1 with len 8: 0x100000
the result is: 1193144
(nemu) p $eax
[src/monitor/debug/expr.c,116,make_token] match rules[12] = "\\$[eE][0-9a-zA-Z]{2}"
at position 0 with len 4: $eax
the result is: 426098000

```

3.3 带有负数的递归求值

负数可解释为单目运算，例如-1 解释为 $-(-1)=1$ ，因此，对于带有负数的表达式时，只需要把-1 转化为 0-1，即可继续接下来的运算。

首先，需要判断 op 是否为单目运算符。若 $l \leq op_index - 1$ ，说明单目运算符是短表达式中的第一个 token，只需要令 $val1=0$ 即可；否则，再判断 op 是否为“+”或“-”，若不是，则正常递归运算。相关代码如下所示。

```

1 uint32_t val1=0;
2     if(l<=op_index-1)
3         val1=eval(l,op_index-1,success);
4     else
5         if(tokens[op_index].type!='+'&&tokens[op_index].type!='-')
6             val1=eval(l,op_index-1,success);

```

在 NEMU 中的测试结果如下所示。

```

(nemu) p ++++-----1
[src/monitor/debug/expr.c,116,make_token] match rules[1] = "+" at position 0 wi
th len 1: +
[src/monitor/debug/expr.c,116,make_token] match rules[1] = "+" at position 1 wi
th len 1: +
[src/monitor/debug/expr.c,116,make_token] match rules[1] = "+" at position 2 wi
th len 1: +
[src/monitor/debug/expr.c,116,make_token] match rules[1] = "+" at position 3 wi
th len 1: +
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 4 wi
th len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 5 wi
th len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 6 wi
th len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 7 wi
th len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[2] = "-" at position 8 wi
th len 1: -
[src/monitor/debug/expr.c,116,make_token] match rules[14] = "[0-9]([1-9][0-9]*)
" at position 9 with len 1: 1
the result is: -1

```

3.4 更复杂的表达式求值

实现 &、|、==、!=、<=、&&、|| 等功能。

修改规则如下所示：

```

1 {" +", TK_NOTYPE}, // spaces
2 {"\\+", '+'}, // plus
3 {"\\-", '-'}, // minus
4 {"\\*", '*'}, // multi/getval
5 {"\\/", '/'}, // div
6 {"<=", TK_LE}, // less than equal
7 {"==", TK_EQ}, // equal
8 {"!=", TK_NEQ}, // not equal
9 {"\\|", TK_OR}, // calc-or
10 {"&", TK_AND}, // calc-and
11 {"!", TK_NOT}, // log-not
12 {"\\|\\|\\|", TK_LOR}, // log-or
13 {"&&", TK_LAND}, // log-and

```

在 make_tokens() 中新加入以下 case。由于结构相似，因此只给出一段代码作为示例。

```

1     case TK_LE:
2     case TK_EQ:
3     case TK_NEQ: {
4         tokens[nr_token].type=rules[i].token_type;

```

```

5         tokens[nr_token].preference=OP_EQ;
6         ++nr_token;
7         break;
8     }

```

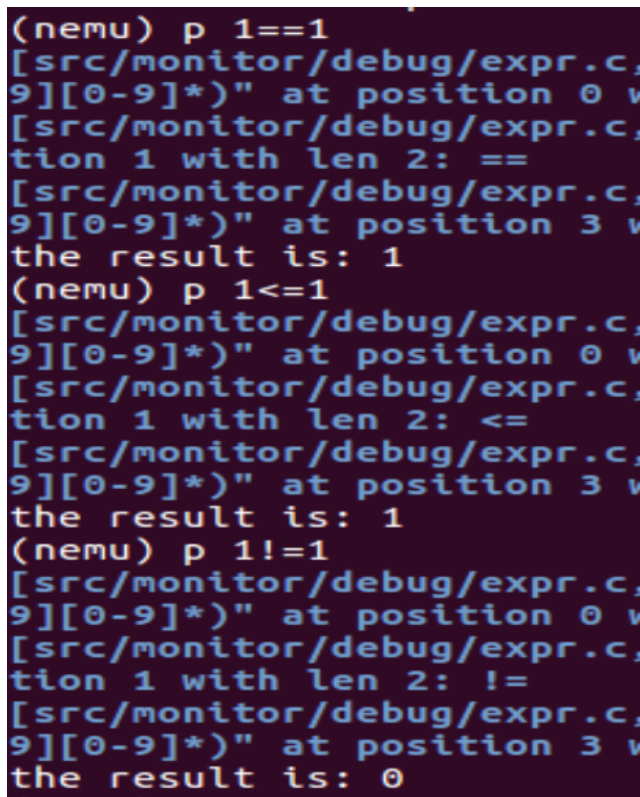
最后，在 eval() 函数中进行补充即可。

```

1     case TK_LE:
2         return val1<=val2;
3     case TK_EQ:
4         return val1==val2;
5     case TK_NEQ:
6         return val1!=val2;
7     case TK_LAND:
8         return val1&&val2;
9     case TK_LOR:
10        return val1||val2;
11    case TK_AND:
12        return val1&val2;
13    case TK_OR:
14        return val1|val2;

```

在 gdb 中进行测试，测试结果如下所示。

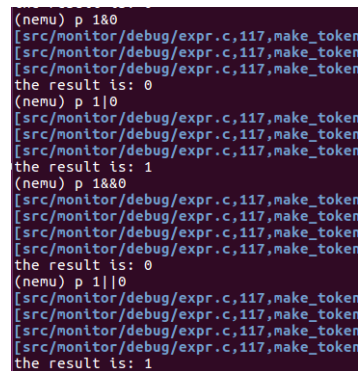


```

(nemu) p 1==1
[src/monitor/debug/expr.c,9][0-9]*" at position 0 w
[src/monitor/debug/expr.c,9][0-9]*" at position 3 w
tion 1 with len 2: ==
[src/monitor/debug/expr.c,9][0-9]*" at position 3 w
the result is: 1
(nemu) p 1<=1
[src/monitor/debug/expr.c,9][0-9]*" at position 0 w
[src/monitor/debug/expr.c,9][0-9]*" at position 3 w
tion 1 with len 2: <=
[src/monitor/debug/expr.c,9][0-9]*" at position 3 w
the result is: 1
(nemu) p 1!=1
[src/monitor/debug/expr.c,9][0-9]*" at position 0 w
[src/monitor/debug/expr.c,9][0-9]*" at position 3 w
tion 1 with len 2: !=
[src/monitor/debug/expr.c,9][0-9]*" at position 3 w
the result is: 0

```

(a)



```

(nemu) p 1&0
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
the result is: 0
(nemu) p 1||0
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
the result is: 1
(nemu) p 1&0
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
the result is: 0
(nemu) p 1||0
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
[src/monitor/debug/expr.c,117,make_token
the result is: 1

```

(b)

4 阶段三

监视点的功能是监视一个表达式的值何时发生变化，当值发生变化时，程序暂停。简易调试器允许用户同时设置多个监视点，删除监视点，打印监视点信息等。因此，需要在本阶段实现监视点。

监视点框架已给出。而由于监视点需要监视表达式的值，因此需要增加两个成员变量，如下所示。

- char expr[50]; //表达式本身，用于后续再次计算
- int value; //表达式的值

4.1 实现监视点池的管理

为了使用监视点池，需要编写 new_wp() 和 free_wp() 两个函数。其中，new_wp() 从 free_ 链表中返回一个空闲的监视点结构。相关代码如下所示：首先检查 free_ 链表是否为空，为空则无法分配新监视点。若不为空，则从表头中获取一个监视点。

```
1 WP* new_wp() {
2     if (free_ == NULL) {
3         printf("no free watch point!");
4         assert (free_ != NULL);
5     }
6     WP * wp = free_;
7     free_ = free_ -> next;
8     wp -> next = head;
9     head = wp;
10    return wp;
11 }
```

free_wp() 将 wp 归还到 free_ 链表中，相关代码如下所示。

由于 head 管理使用的监视点，因此，只需要遍历 head，找到待删的监视点，将其移除，并链入 free_ 链表中即可。

```
1 void free_wp(WP *wp) {
2     WP *cur = head;
3     WP *pre = NULL;
4     while (cur) {
5         if (cur == wp) {
6             if (pre)
7                 pre -> next = cur -> next;
8             else
9                 head = cur -> next;
10            cur -> next = free_;
11            free_ = cur;
12            break;
13        }
14        pre = cur;
15        cur = cur -> next;
16    }
17 }
```

4.2 问题 1: static 的含义及作用

【Q】：框架代码中定义 wp_pool 等变量的时候使用了关键字 static, static 在此处的含义是什么？为什么要在此处使用它？

【A】：在这段代码中，关键字 static 被用来声明变量为静态变量。静态变量在程序运行期间只会被初始化一次，在整个程序的生命周期内都会保留其数值，不会因为函数的调用结束而被销毁。

因此，static 在此处的含义是静态，在此使用的目的是：使用 static 关键字声明 wp_pool、head 和 free_ 变量是为了将它们定义为静态变量，以确保它们在整个程序运行期间都可以被访问和使用，并且只会被初始化一次。这样可以保证这些变量在程序的不同部分都能够共享相同的数据，而不会因为函数的调用结束而失效。

4.3 实现监视点

4.3.1 打印监视点信息

命令	格式	使用举例	说明
打印程序状态	info SUBCMD	info w	打印监视点信息

在阶段一已实现打印寄存器状态 (info r)，在这里主要实现打印监视点信息 (info w)。首先需要在 watchpoint.c 中实现打印函数，即打印监视点序号、表达式、表达式当前值。打印函数代码如下所示。

```

1 void print_wp(){
2     WP *cur=head;
3     if(cur==NULL){
4         printf("No watchpoints!\n");
5         return;
6     }
7     while(cur){
8         printf("No:%d, Expr:%s, Val:%d\n",cur->NO,cur->expr,cur->value);
9         cur=cur->next;
10    }
11 }

```

之后，在 ui.c 的 cmd_info 函数中新增“w”条件分支，调用 print_wp 函数即可。

```

1 else if(strcmp(arg,"w")==0)
2     print_wp();

```

4.3.2 设置监视点

命令	格式	使用举例	说明
设置监视点	w EXPR	w *0x2000	当表达式 EXPR 的值发生变化时，暂停程序执行

在 ui.c 中新增 cmd_w() 函数，相关代码如下所示。由于 SUBCMD 为表达式，因此调用在阶段二实现的计算表达式函数 expr(char[],bool)，计算结果存入监视点的 value 变量，表达式本身存入监视点的 expr 变量。

```

1 {"w","Set watchpoint at expr's value",cmd_w},
2 static int cmd_w(char *args{
3     char *arg =strtok(NULL, "");
4     if(arg==NULL){
5         printf("Please input expression\n");
6         return 0;
7     }
8     bool success=false;
9     uint32_t result=expr(arg, &success);
10    if(!success){
11        printf("Your expression is erroneous!\n");
12        return 0;
13    }
14
15    WP *wp=new_wp();
16    wp->value=result;
17    int i;
18    for(i=0;arg[i];i++)
19        wp->expr[i]=arg[i];
20    wp->expr[i]='\0';
21    printf("Set watchpoint successfully!\n");
22    return 0;
23 }

```

4.3.3 删除监视点

命令	格式	使用举例	说明
删除监视点	d N	d 2	删除序号为 N 的监视点

在 ui.c 中新增 cmd_d() 函数，用于删除监视点。相关代码如下所示。

```

1 {"d","Delete watchpoint",cmd_d},
2 static int cmd_d(char *args{
3     char *arg =strtok(NULL, "");
4     if(arg==NULL){
5         printf("Please input the wp's NO\n");
6         return 0;
7     }
8     int no = atoi(arg);
9     WP *wp=search_wp(no);
10    if(wp)
11        free_wp(wp);
12    else
13        printf("No such watchpoint!\n");
14    return 0;
15 }

```

由于 SUBCMD 为待删监视点代码，而上一节完成的 free_wp() 函数的参数为 WP*，因此需要新增查找函数 search_wp()，用于在监视点链表中根据监视点序号查找待删监视点。代码如下所示。

```

1 WP *search_wp(int NO){
2     WP *cur = head;
3     while(cur){
4         if(cur->NO == NO)
5             return cur;
6         cur = cur -> next;
7     }
8     return NULL;
9 }

```

4.3.4 实现暂停操作

至此，我们设置的监视点并没有起到实际作用，必须要添加检测函数，并实现暂停操作才能让监视点生效。

实现监视点检查函数：遍历所有监视点，重新计算表达式，若表达式的值发生变化，则更新新值，并记录变化情况。当至少一个监视点的值改变，则暂停程序。检查监视点的函数如下所示：

```

1 bool check_watchpoints_value( ){
2     WP *cur=head;
3     bool flag=false ,success=true;
4     while(cur){
5         uint32_t result=expr(cur->expr,&success);
6         if(!success){
7             printf("Calculate expression fail/n");
8             continue;
9         }
10        if(result != cur->value){
11            flag=true;
12            printf("No:%d, Expr:%s, Val:%d, New Val:%d\n", cur->NO, cur->expr, cur->value,
13                result);
14            cur->value=result;
15        }
16        cur=cur->next;
17    }
18    return flag;
19 }

```

在 cpu_exec() 函数的 #ifdef DEBUG 处增加暂停操作。每执行一条指令即检查一次监视点状态，若存在监视点的值发生变化，修改 nemu_state 为 NEMU_STOP。代码如下所示：

```

1 #ifdef DEBUG
2     if(check_watchpoints_value())
3         nemu_state=NEMU_STOP;

```


【Q】: ModR/M 字节是什么?

【A】: Page241 17.2.1 ModR/M and SIB Bytes

【Q】: mov 指令的具体格式是怎么样的?

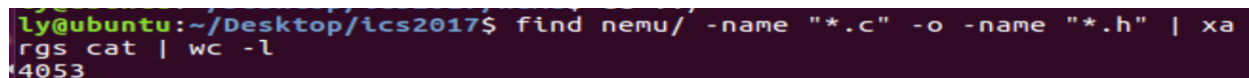
【A】: Page345-351

5.2 问题 2: shell 命令统计代码行数

【Q】: shell 命令完成 PA1 的内容之后,nemu/目录下的所有.c 和.h 和文件总共有多少行代码? 你是使用什么命令得到这个结果的?

【A】: 共 4053 行, 使用的命令为

```
1 find nemu/ -name "*.c" -o -name "*.h" | xargs cat | wc -l
```



```
ly@ubuntu:~/Desktop/ics2017$ find nemu/ -name "*.c" -o -name "*.h" | xargs cat | wc -l
4053
```

【Q】: 和框架代码相比, 你在 PA1 中编写了多少行代码?(Hint: 目前 2017 分支中记录的正好是做 PA1 之前的状态, 思考一下应该如何回到”过去”?)

【A】: 566 行。使用 git checkout pa0 指令即可来到 pa0 分支, 再次使用统计指令, 即可得知框架代码行数 (3487)。PA1 分支与 pa0 分支代码数相减即可得到在 PA1 中编写的代码数。

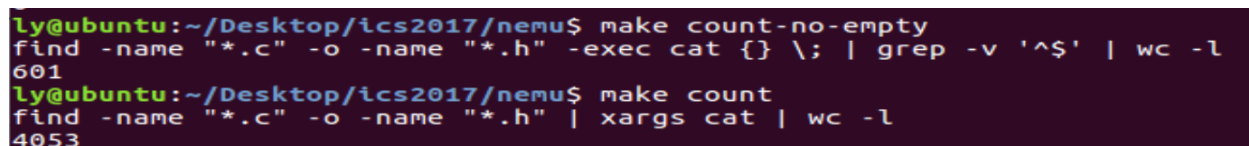
【Q】: 你可以把这条命令写入 Makefile 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 make count 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外,nemu/目录下的所有.c 和.h 文件总共有多少行代码?

【A】: 在 Makefile 末尾补充如下内容。

```
1 count:
2     find -name "*.c" -o -name "*.h" | xargs cat | wc -l
```

若要除去空行, 则将命令改为

```
1 find nemu/ -name "*.c" -o -name "*.h" -exec cat {} \; | grep -v '^$' | wc -l
```



```
ly@ubuntu:~/Desktop/ics2017/nemu$ make count-no-empty
find -name "*.c" -o -name "*.h" -exec cat {} \; | grep -v '^$' | wc -l
601
ly@ubuntu:~/Desktop/ics2017/nemu$ make count
find -name "*.c" -o -name "*.h" | xargs cat | wc -l
4053
```

5.3 问题 3: gcc 编译选项-Wall 与-Werror 的作用

【Q】: 使用 man 打开工程目录下的 Makefile 文件, 你会在 CFLAGS 变量中看到 gcc 的一些编译选项. 请解释 gcc 中的-Wall 和-Werror 有什么作用? 为什么要使用-Wall 和-Werror?

【A】: -Wall 和-Werror 是 GCC 编译器的编译选项，它们的作用如下：

- -Wall：这个选项表示开启所有警告信息。编译器会检查代码中的潜在问题，并给出警告信息。这些警告可能包括未使用的变量、未初始化的变量、类型不匹配等。开启-Wall 选项有助于发现潜在的代码问题，提高代码质量。
- -Werror：这个选项表示将所有警告视为错误。一旦编译器发出任何警告，编译过程就会被终止，并将警告视为错误。这迫使开发者解决所有警告，确保代码的严谨性和可靠性。

使用-Wall 和-Werror 的目的在于：使用-Wall 可以帮助开发者发现代码中可能存在的问题，提前进行修复，避免潜在的 bug；而使用-Werror 可以确保代码在编译时不会产生任何警告，强制开发者保持高质量的代码风格，防止忽略警告而导致潜在的问题。

综上所述，综合使用-Wall 和-Werror 有助于提高代码质量、可读性和可维护性。