预备工作 2 定义我的编译器&汇编编程

一、定义我的编译器

1. 我的编译器所支持的 C 语言特性

函数：函数可以带参数也可以不带参数，参数的类型可以是 int 或者数组类型；函数可以返回 int 类型的值，或者不返回值(即声明为 void 类型)。当参数为 int 时，按值传递；而参数为数组类型时，实际传递的是数组的起始地址，并且形参只有第一维的长度可以空缺。函数体由若干变量声明和语句组成。

变量/常量声明：可以在一个变量/常量声明语句中声明多个变量或常量，声明时可以带初始化表达式。所有变量/常量要求先定义再使用。在函数外声明的为全局变量/常量，在函数内声明的为局部变量/常量。

语句：语句包括赋值语句、表达式语句(表达式可以为空)、语句块、if 语句、while 语句、switch 语句、case 语句、return 语句。语句块中可以包含若干变量声明和语句。

表达式：支持基本的算术运算（+、-、*、/、%）、关系运算（==、!=、<、>、<=、>=）和逻辑运算（!、&&、||），非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。算符的优先级和结合性以及计算规则与 **C** 语言一致。

2.上下文无关文法描述我的 C 语言子集

程序→声明列表
声明列表→声明列表 | 声明
声明→变量声明 | 常量声明 |函数声明 | 函数定义
常量声明→'const' 数据类型 常数定义 { ',' 常数定义}';'
常数定义→ID {['常量表达式']'}'='常量初值
常量初值→常量表达式 | '{' [常量初值{',' 常量初值}] '}'
变量声明→数据类型 变量列表
变量列表→变量列表 变量 | 变量
变量→ID | ID[表达式]
数据类型→void | int
函数定义→变量类型 ID（参数列表）复合语句
参数列表→参数列表 参数 | 参数
参数→数据类型 | 数据类型 ID | 数据类型 ID [ ]|数据类型 ID [表达式] | 数据类型 ID [ ][ ]|

数据类型 ID[表达式][表达式]
复合语句→{局部说明 语句列表}
局部说明→局部说明 变量说明 | 变量说明
语句列表→语句列表 语句 | 语句
语句→表达式语句 | 循环语句 | 返回语句 | 选择语句
选择语句→if(表达式) 语句 |if（表达式）语句 else（表达式）语句 |switch（表达式）语句
标志语句→case 表达式：语句 | default 表达式：语句
循环语句→while（表达式）语句 | do 语句 while（表达式）
返回语句→return | return 表达式
表达式语句→表达式 ； | ；
表达式→加减表达式
条件表达式→逻辑或表达式
左值表达式→ID { '['表达式']' }
基本表达式→'(' 表达式 ')' | 左值表达式 | 数值
数值→整型常量
一元表达式→基本表达式 | ID'(' [函数实参表] ')' | 单目运算符 一元表达式
单目运算符→'+' | '-' | '!'
函数实参表→表达式 {',' 表达式}
乘除模表达式→ 一元表达式 | 乘除模表达式 ('*' | '/' | '%') 一元表达式
加减表达式→ 乘除模表达式 | 加减表达式 ('+' | '−') 乘除模表达式
关系表达式→ 加减表达式 | 关系表达式 ('<' | '>' | '<=' | '>=') 加减表达式
相等性表达式→ 关系表达式 | 相等性表达式 ('==' | '!=') 关系表达式
逻辑与表达式→相等性表达式 | 逻辑与表达式 '&&' 相等性表达式
逻辑或表达式→ 逻辑与表达式 | 逻辑或表达式 '||' 逻辑与表达式
常量表达式→ 加减表达式

3.终结符特征

（1）标识符 ID
ID → ID ID-nondigit | ID ID-digit | ID-nondigit
其中 ID-nondigit 为 26 个英文字母的大小写组成的集合
ID-digit 为数字 0-9
（2）注释
单行注释：以序列'//' 开始，直到换行结束，不包括换行符。
多行注释：以序列 '/*' 开始，直到第一次出现 '*/' 时结束，包括结束处'*/'。
（3）数值常量
整型常量→整型常量 | 整型常量 digit
digit 为数字 0-9

二、汇编编程

尝试对 lab_01 中的阶乘程序进行汇编编程。
源码如下：

```
#include<stdio.h>
int main()
{
   int i, n, f;
   scanf(“%d”, &n);
   i = 2;
   f = 1;
   while(i<=n)
{
   f = f * i;
   i = i +1;
}
printf(“%d\n”, f);
return;
}
```

0. 编码与编译

自己编写的 *x86_64* 汇编代码如下：

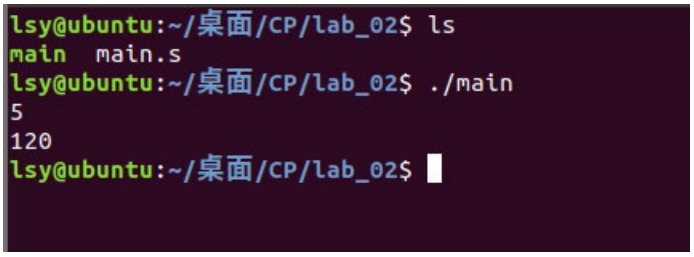
```
.section  .rodata
.LC0:
     .string "%d"
.LC1:
     .string "%d\n"
#main function
.text
.globl main
.type  main, @function
main:
.LFB23:
     #给栈分配 24 字节的空间
     subq  $24, %rsp
     #设置返回值寄存器为 0
     movl  $0,%eax
     leaq  4(%rsp),%rsi
     #将 LC0 的有效地址加载到 rdi 寄存器
     leaq  .LC0(%rip),%rdi
```

```
19      call  __isoc99_scanf@PLT
20      movl $1,%edx
21      movl $2,%ecx
22      jmp .L2
23  .L3:
24      imull %ecx,%edx
25      addl  $1, %ecx
26  .L2:
27      cmpl %ecx, 4(%rsp)
28      jge  .L3
29      leaq .LC1(%rip),%rsi
30      call __printf_chk@PLT
31      addq $24, %rsp
32      ret
33  .section        .note.GNU-stack,"",@progbits
```

编写完成后用 gcc 生成可执行文件进行测试

在终端输入命令:

```
gcc main.s -o main
```



输入 5 得到正确结果，编译成功!

源程序的 C 代码保存在 test1.c 文件中，用 gcc -S 命令自动生成源程序的汇编代码与自己写的 main.s 进行比较，使用文本比较工具 *colordiff*
输入命令:

```
colordiff -y -B -b main.s test1.s
```

-B 参数表示忽略空行造成的不同，-b 表示忽略空格造成的不同，-y 参数可以让输出结果将屏幕分为两半方便我们查看。结果如下:

1. main 函数部分：

屏幕左边是自己写的汇编代码，右边为系统生成的代码。

首先可以明显看到整体上系统生成的代码比自己写的要长不少，屏幕右边中绿色的部分都是比自己写的代码多出的行。除去一些以“*.cfi*”开头的系统调用命令，系统生成的汇编代码中也多出了很多对寄存器（特别是堆栈指针寄存器）的操作。**感觉好像如果执行的命令更多会降低这个程序的效率，但是编译器为什么要这么做呢？**这其中一定有原因，我开始尝试弄清楚为什么编译器生成的代码会多出这么多东西。

（1）首先发现的一个细节的不同是对于给返回值寄存器%eax 赋值 0 的操作

我直接使用了 `movl $0, %eax` 命令，而右边用的是 `xorl %eax, %eax`

同样是给 eax 寄存器赋 0，但让它与自己进行异或运算比用 mov 指令赋值效率更高

这个发现启发了我，或许两个汇编代码的差异是编译器对源程序进行了**优化**导致的结果

（2）另外注意到两行特殊的代码：

```
movq %fs:40, %rax
movq %rax, -8(%rbp)
```

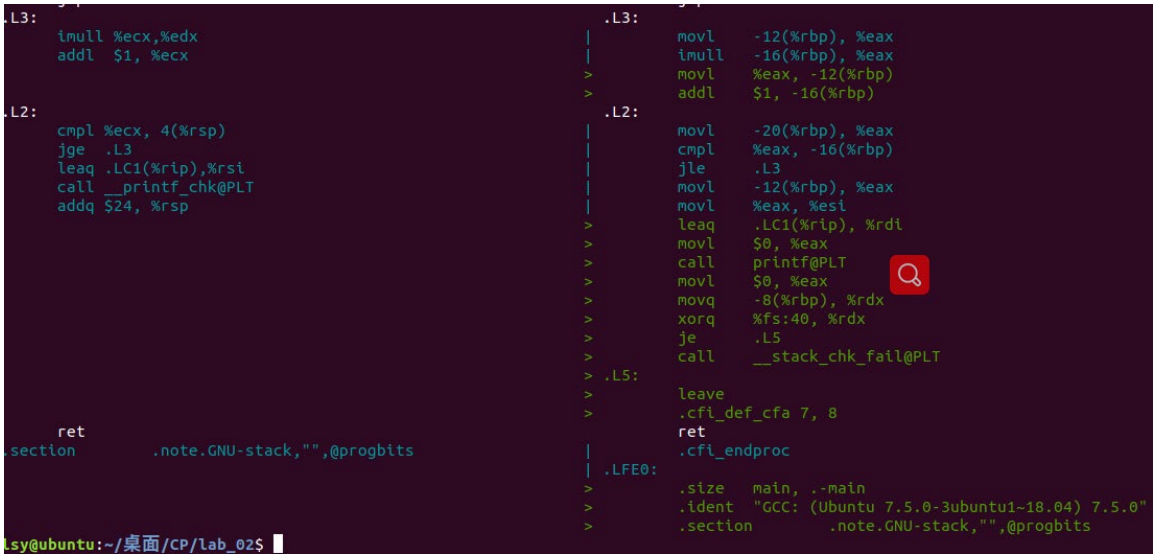
查阅资料（《***深入理解计算机系统***》3.10.4 节“对抗缓冲区溢出攻击”）得知，这两行代码包括后面许多对寄存器和堆栈指针的操作都是为了防止缓冲区溢出和对抗缓冲区溢出攻击。

书中原文描述如下：**这是一种“栈保护者（STACK PROTECTOR）”机制，来检测缓冲区越界。其思想是在栈帧中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀（CANARY）值，也称为哨兵值（GUARD VALUE），是在程序每次运行时随机产生的，因此，攻击者没有简单的办法能够知道它是什么。在恢复寄存器状态的从函数返回之前，程序检查这个金丝雀值是否被某个函数的某个操作或者该函数调用的某个函数的某个操作改变了。如**

果是的，那么程序异常中止。这种栈保护只会带来很小的性能损失。[1]

另外，在对局部变量进行赋值时，由于代码量很小，所以我直接用了 edx 和 ecx 寄存器来进行赋值，而右边代码中是将值给到栈中，这里算是我仗着代码量小用到的寄存器不多然后偷懒了。

2. while 循环部分

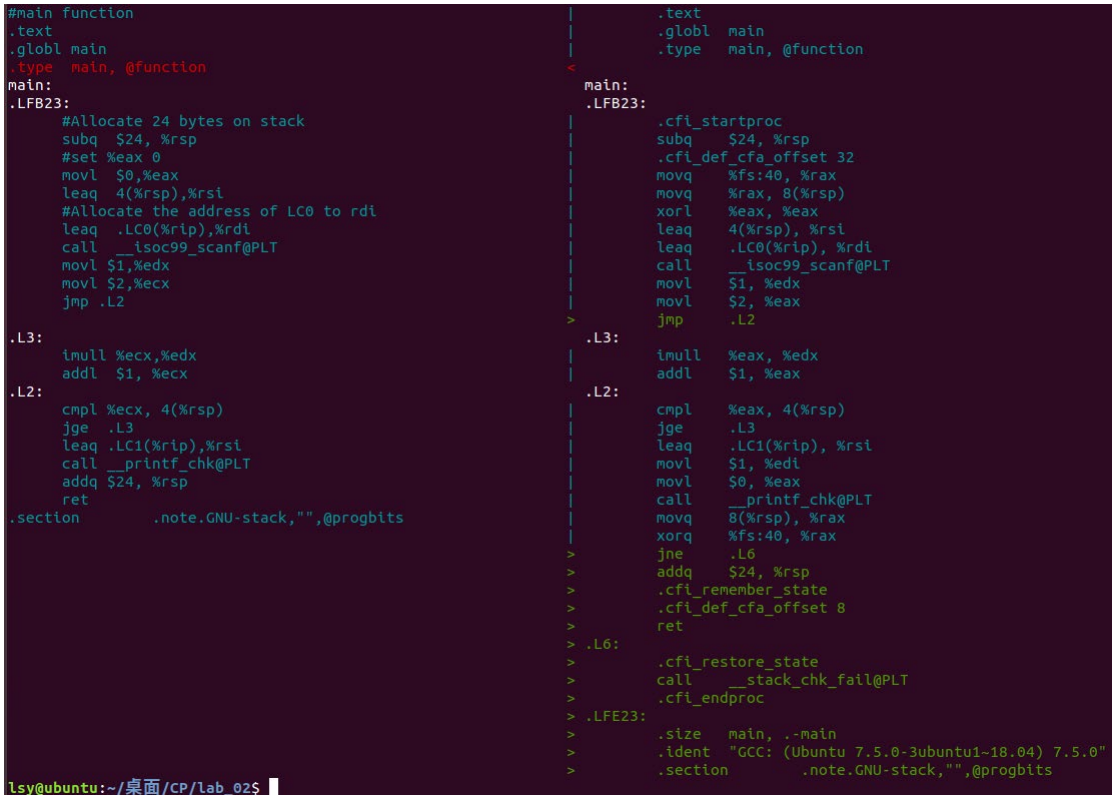


这部分代码的思路整体一致，多出的部分仍然是栈保护机制的内容，其余部分主要区别也还是由于我直接使用寄存器运算而右边代码是调用堆栈进行运算导致的。

3. 取消优化后的代码比较

在用 gcc 生成源程序汇编代码时加上参数-O0，不进行任何优化生成汇编代码，再与自己的代码进行比较。

[1] 《*深入理解计算机系统*》Randal E. Bryant、David R. O'Hallaron 机械工业出版社 P199



这次可以发现，如果去掉*.cfi*调用命令，代码长度没有明显差别了。但即使加入了参数O0，栈保护机制在右边代码中仍然存在，其余差别都比较细微，无关紧要。不过有一点被我注意到，在进行常量赋值时，右边代码直接使用了 eax 寄存器，虽然这个寄存器本来是函数返回值寄存器，但由于这个 main 函数是 void 类型，所以 eax 就被它“挪作它用”了，但这样做一个个小小的好处是，由于前面进行过给 eax 赋 0 的操作，所以接下来 CPU 再访问刚刚访问过的寄存器会比访问新的寄存器要快一点。当然这一点点性能的提升在这个小程序中体现不出来。