实验题目：lab0，操作系统的编程基础

实验环境

Description: Ubuntu 16.04 LTS

Release: 16.04

Codename: xenial

git version 2.7.4

GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1

安装linux环境，并安装gcc和gdb。

仓库地址:git clone https://github.com/Chasssser/Mytest

1. 预备知识

GCC扩展内联汇编格式(asm（语句：输出寄存器，输入寄存器，会被破坏的寄存器）)

asm [volatile] ( Assembler Template

: Output Operands

[ : Input Operands

[ : Clobbers ] ])

通用限定符

a eax

b ebx

c ecx

d edx

S esi

D edi

I 常数值，(0 - 31)

q,r 动态分配的寄存器

g eax,ebx,ecx,edx或内存变量

A 把eax和edx合成一个64位的寄存器(use long longs)

AT&T基本语法

\* 寄存器命名原则

AT&T: %eax Intel: eax

\* 源/目的操作数顺序

AT&T: movl %eax, %ebx Intel: mov ebx, eax

\* 常数/立即数的格式

AT&T: movl $\_value, %ebx Intel: mov eax, \_value

把value的地址放入eax寄存器

AT&T: movl $0xd00d, %ebx Intel: mov ebx, 0xd00d

\* 操作数长度标识

AT&T: movw %ax, %bx Intel: mov bx, ax

\* 寻址方式

AT&T: immed32(basepointer, indexpointer, indexscale)

Intel: [basepointer + indexpointer × indexscale + imm32)

常用寄存器(来自wiki百科)(https://en.wikipedia.org/wiki/X86#x86-64)

### **Purpose**

Although the main registers (with the exception of the instruction pointer) are "general-purpose" in the 32-bit and 64-bit versions of the instruction set and can be used for anything, it was originally envisioned that they be used for the following purposes:

* AL/AH/AX/EAX/RAX: Accumulator
* BL/BH/BX/EBX/RBX: Base index (for use with arrays)
* CL/CH/CX/ECX/RCX: Counter (for use with loops and strings)
* DL/DH/DX/EDX/RDX: Extend the precision of the accumulator (e.g. combine 32-bit EAX and EDX for 64-bit integer operations in 32-bit code)
* SI/ESI/RSI: *Source index* for [string](https://en.wikipedia.org/wiki/String_(computer_science)" \o "String (computer science)) operations.
* DI/EDI/RDI: *Destination index* for string operations.
* SP/ESP/RSP: Stack pointer for top address of the stack.
* BP/EBP/RBP: Stack base pointer for holding the address of the current [stack frame](https://en.wikipedia.org/wiki/Stack_frame" \o "Stack frame).
* IP/EIP/RIP: Instruction pointer. Holds the [program counter](https://en.wikipedia.org/wiki/Program_counter" \o "Program counter), the address of next instruction.

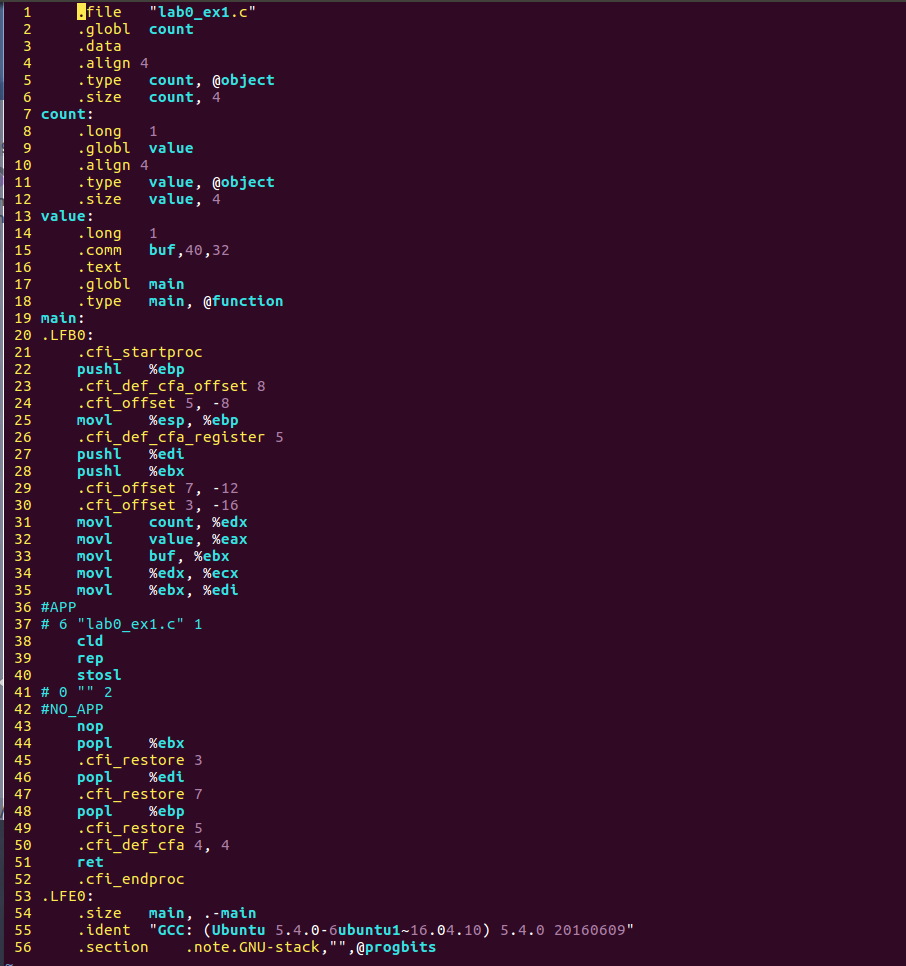
Segment registers:

* CS: Code
* DS: Data
* SS: Stack
* ES: Extra data
* FS: Extra data #2
* GS: Extra data #3

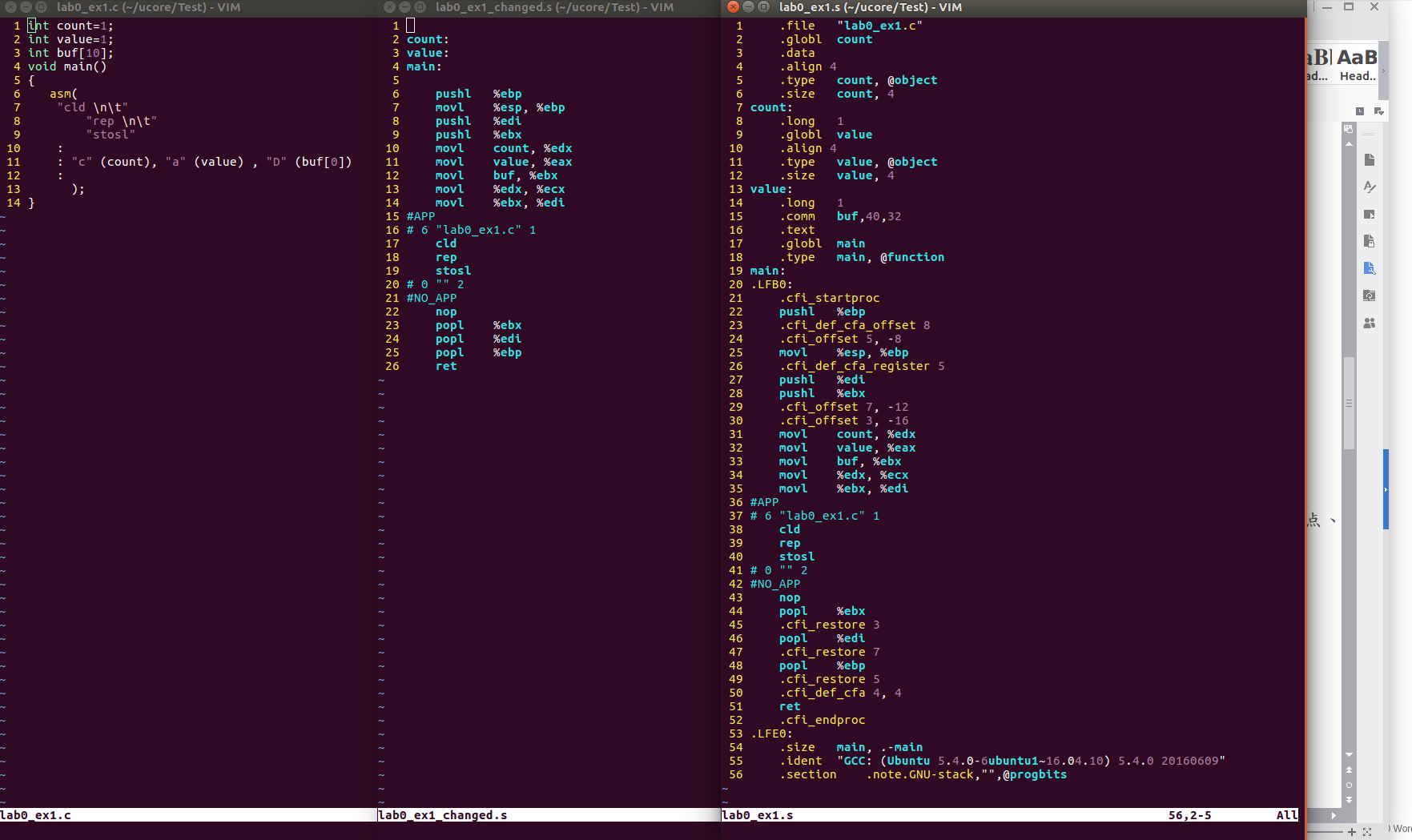
1. 了解汇编

尝试理解下面的命令

$gcc -S -m32 lab0\_ex1.c

接着我们将得到lab0\_ex1.s文件，请写出汇编代码与c代码之间的关系。

其中有很多.开头的代码，属于链接的辅助信息(上图黄色开头部分)，所以删除可以得到汇编代码，如下:.c--.s(changed)--.s



先看.c，里面使用了gcc扩展内联汇编.

"cld\n\t",这是清除了寄存器的方向标记。

"rep\n\t" (因为使用的是GAS编译器，过意要求rep前缀单独占一行，rep前缀是字符串重复操作前缀)

"stosl"(stos有个后缀l(字母)指明它每次移动一个长字)

在上面的.c文件中第10行为空，说明没有指定输出的寄存器。

接下来一行中 : "c" (count), "a" (value), "D" (buf[0])，参考限定符可以知道count值被保存在ecx(for use with loops and strings)中，fill\_value被保存在eax(Accumulator)中，edi(Destination index for string operations)中的是目的地址.

所以这几句话的意思就是向buf地址里写入count个value的值。(个人感觉有点像memset，就是不知道memset会不会也生成内联代码，mark一下，找时间看看O(∩\_∩)O～)

限定符位于引号之内，表达式放在圆括号内，可以告诉编译器需要改变那些寄存器。

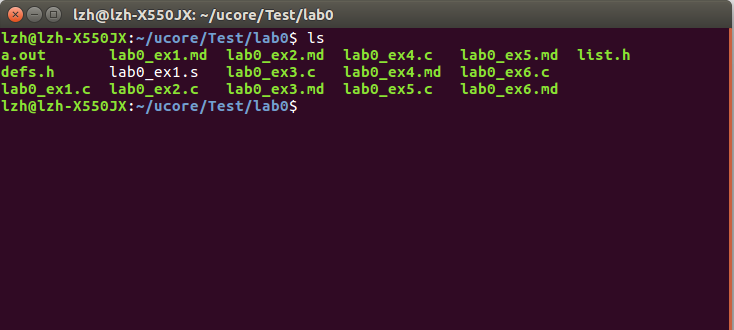
2.用gdb调试

尝试下面的命令，

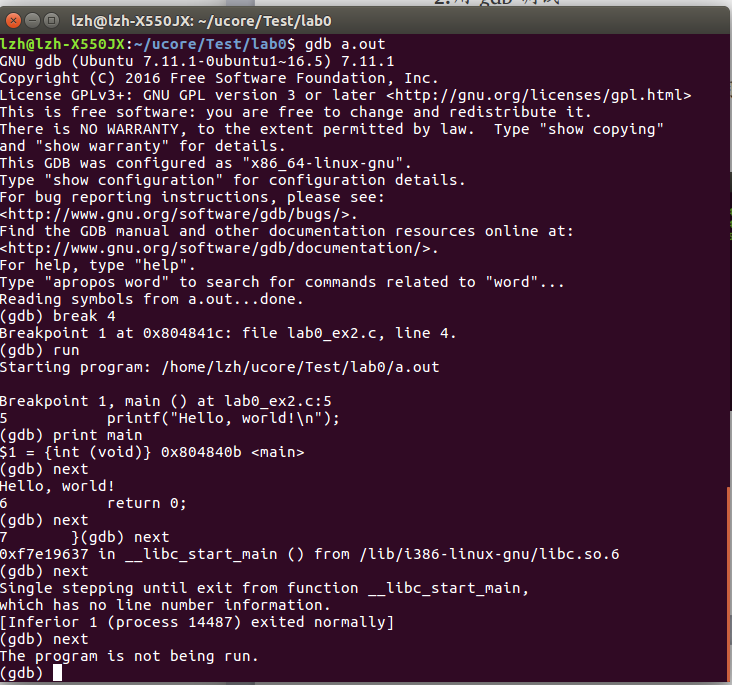
$gcc -g -m32 lab0\_ex2.c

接着我们会得到a.out文件，请用gdb调试，并写出设置断点、单步执行及查看变量的过程。

先得到a.out



然后打断点



3.掌握指针和类型转换相关的Ｃ编程

分析如下代码段，

#include <stdio.h>

#define STS\_IG32 0xE // 32-bit Interrupt Gate

#define STS\_TG32 0xF // 32-bit Trap Gate

typedef unsigned uint32\_t;

#define SETGATE(gate, istrap, sel, off, dpl) { \

(gate).gd\_off\_15\_0 = (uint32\_t)(off) & 0xffff; \

(gate).gd\_ss = (sel); \

(gate).gd\_args = 0; \

(gate).gd\_rsv1 = 0; \

(gate).gd\_type = (istrap) ? STS\_TG32 : STS\_IG32; \

(gate).gd\_s = 0; \

(gate).gd\_dpl = (dpl); \

(gate).gd\_p = 1; \

(gate).gd\_off\_31\_16 = (uint32\_t)(off) >> 16; \

}

/\* Gate descriptors for interrupts and traps \*/

struct gatedesc {

unsigned gd\_off\_15\_0 : 16; // low 16 bits of offset in segment

unsigned gd\_ss : 16; // segment selector

unsigned gd\_args : 5; // # args, 0 for interrupt/trap gates

unsigned gd\_rsv1 : 3; // reserved(should be zero I guess)

unsigned gd\_type : 4; // type(STS\_{TG,IG32,TG32})

unsigned gd\_s : 1; // must be 0 (system)

unsigned gd\_dpl : 2; // descriptor(meaning new) privilege level

unsigned gd\_p : 1; // Present

unsigned gd\_off\_31\_16 : 16; // high bits of offset in segment

};

int

main(void)

{

unsigned before;

unsigned intr;

unsigned after;

struct gatedesc gintr;

intr=8;

before=after=0;

gintr=\*((struct gatedesc \*)&intr);

SETGATE(gintr, 0,1,2,3);

intr=\*(unsigned \*)&(gintr);

printf("intr is 0x%x\n",intr);

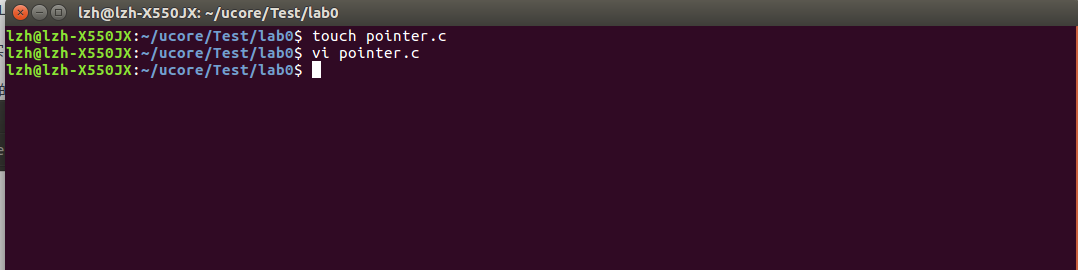
printf("intr is 0x%llx\n", gintr);

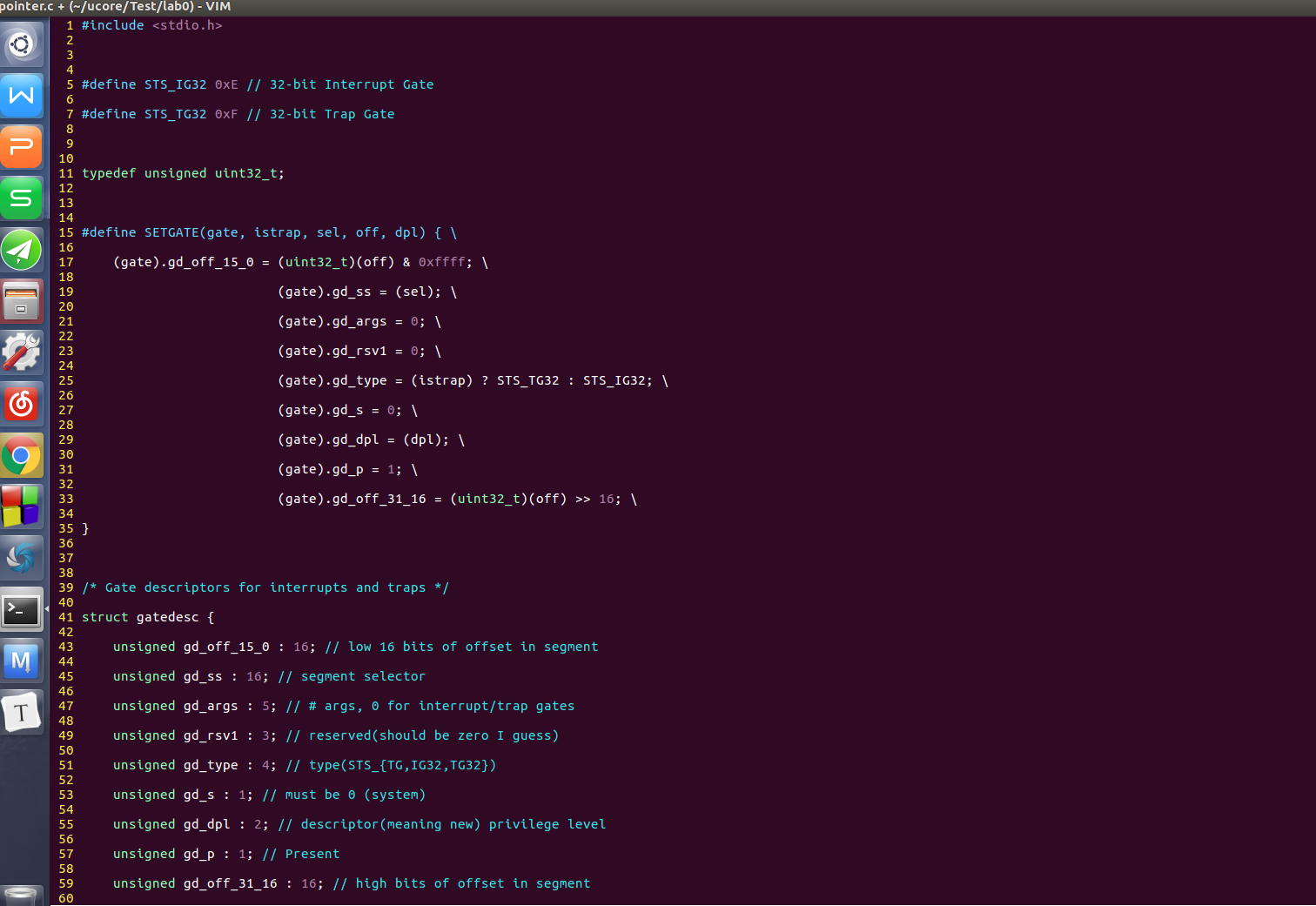
return 0;

}

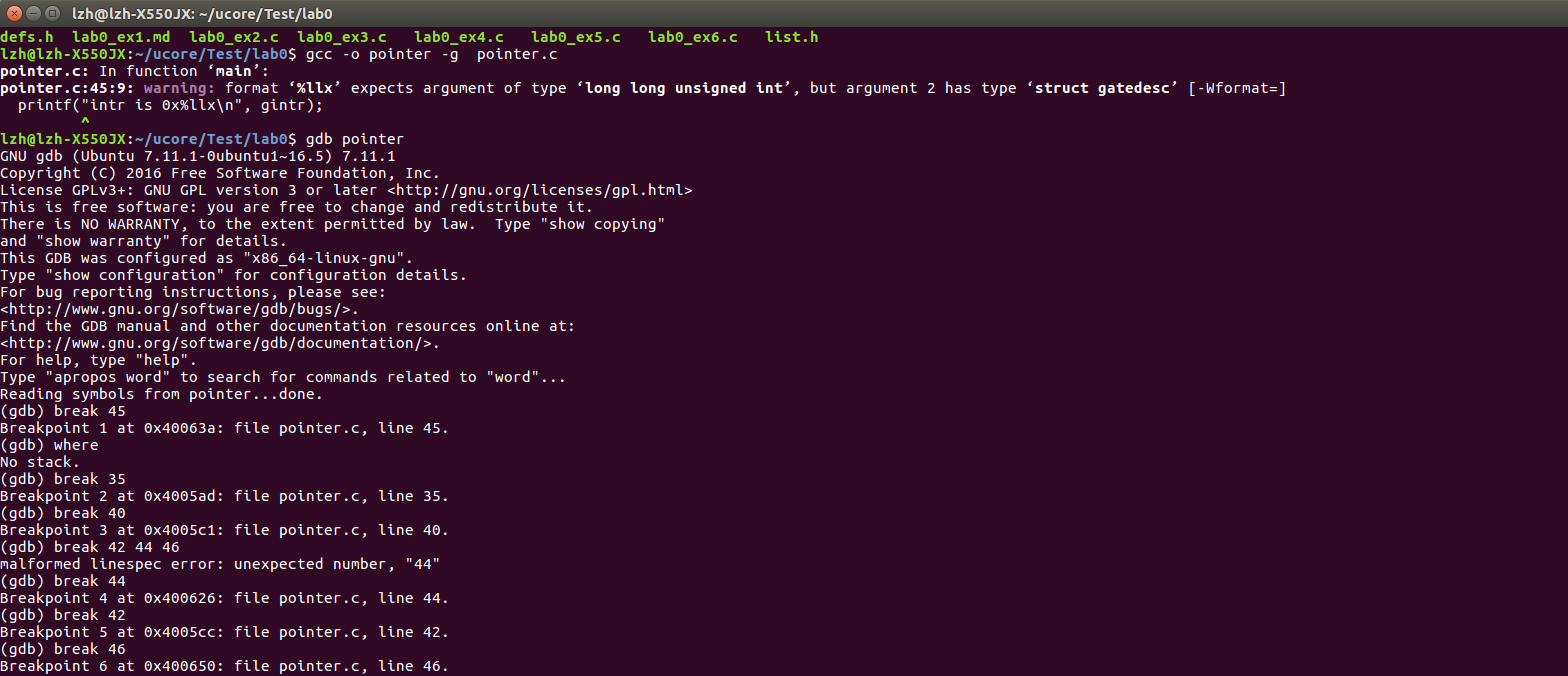
写出gintr和intr的结果，试着编译这段代码，如果遇到错误进行改正，并分析错误原因。

新建pointer.c并将上述代码写入pointer.c文件中





然后使用gcc编译，gdb调试。

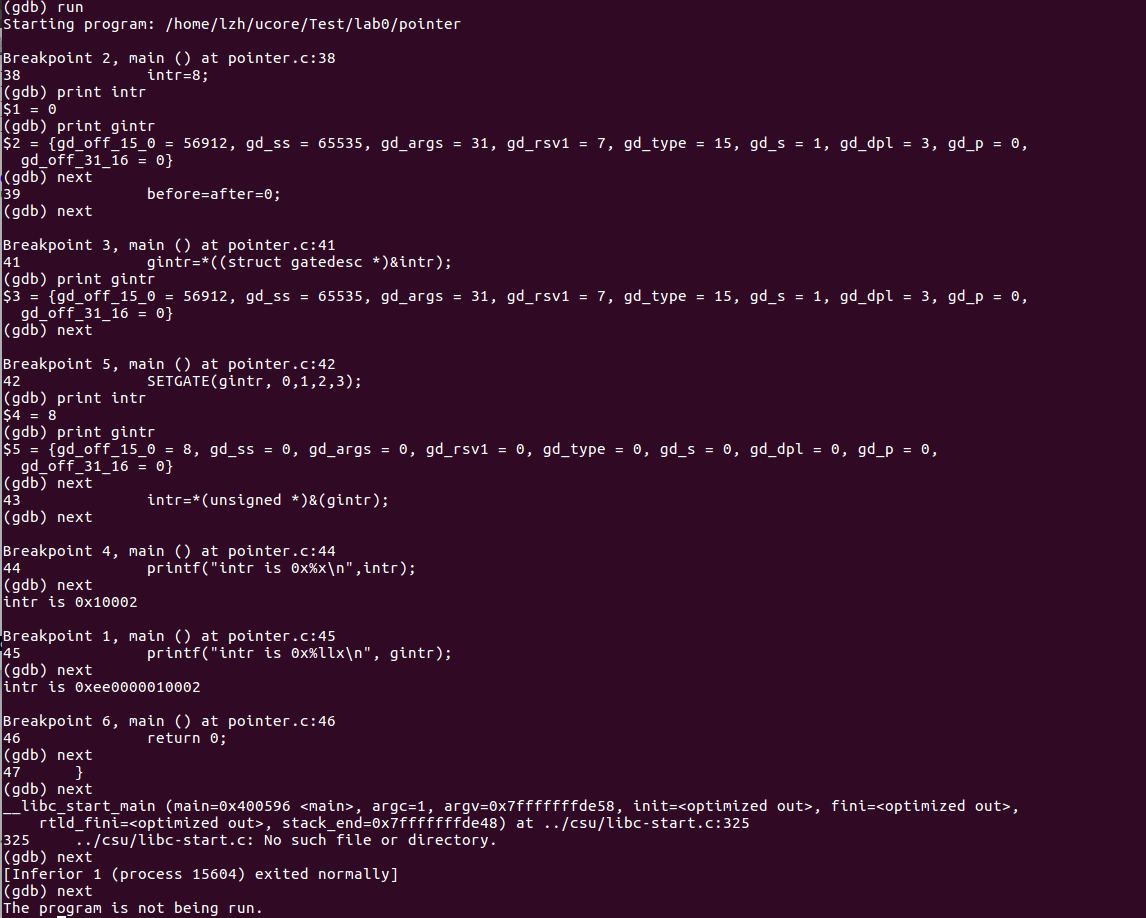


gcc -o pointer -g pointer.c编译

可以看到有个warning在第45行,因为使用了%llx格式来输出struct gatedesc。

然后gdb pointer开始调试

为了测试，先后第45,35,40,44,42和46行打了断点

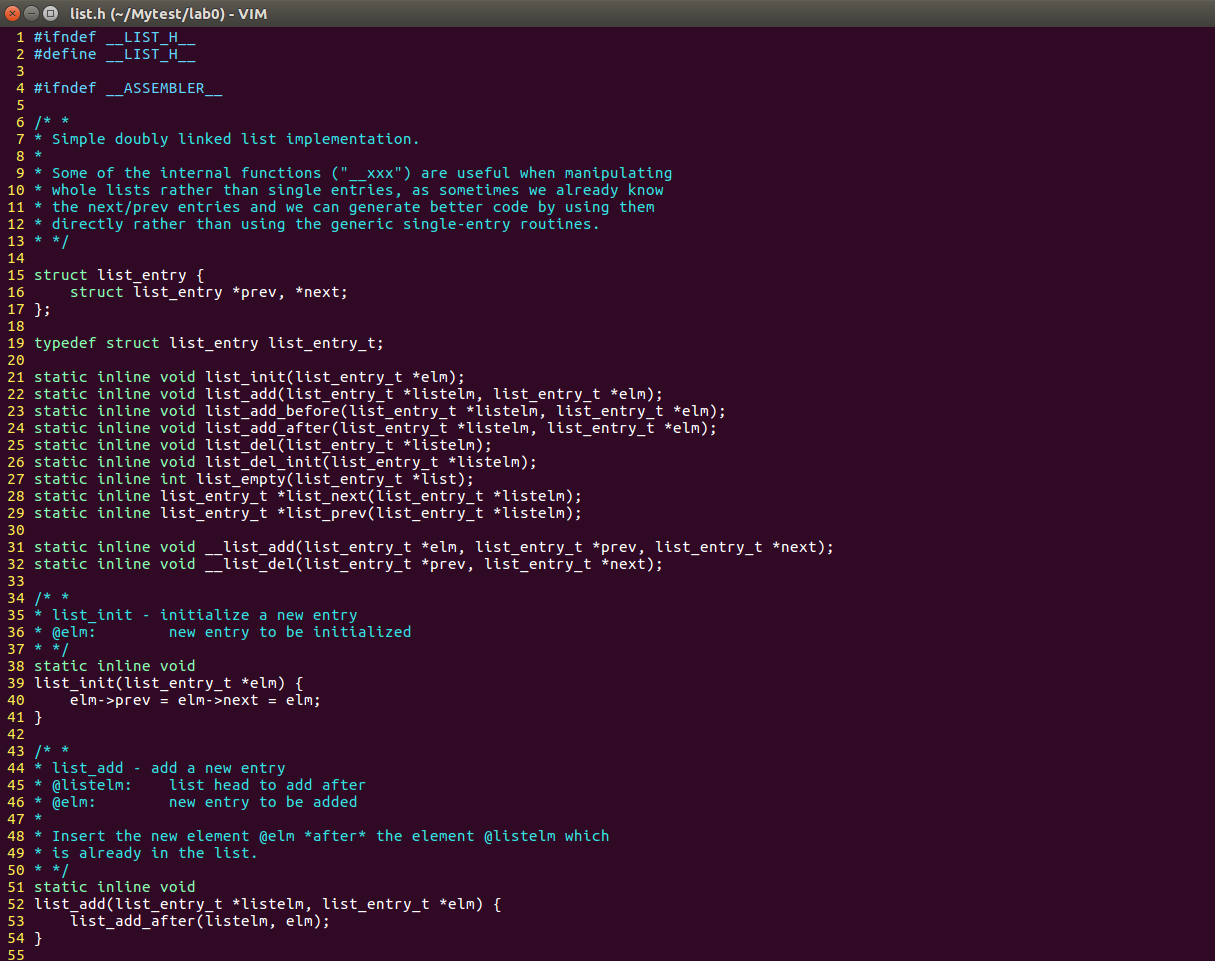


Next重复执行，遇到断点就print intr和gintr，可以看到intr和gintr的值发生变化.

4. 掌握通用链表结构相关的Ｃ编程

查看list.h和lab0\_ex4.c，编写一个程序，利用list.h中的链表结构，将26个英文字母存入链表中，并逆序打印出来。

List.h



Lab0\_ex4.c

