实验题目：lab0，操作系统的编程基础

实验环境

<http://www.shiyanlou.com/courses/221>

安装linux环境，并安装gcc和gdb。

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。

**汇编语言用栈 寄存器 来执行c代码**

**cld \n\t"*//将标志寄存器Flag的方向标志位DF清零。***

**"rep \n\t"*//重复前缀指令***

**"stosl"*//将EAX中的值保存到ES:EDI指向的地址中***

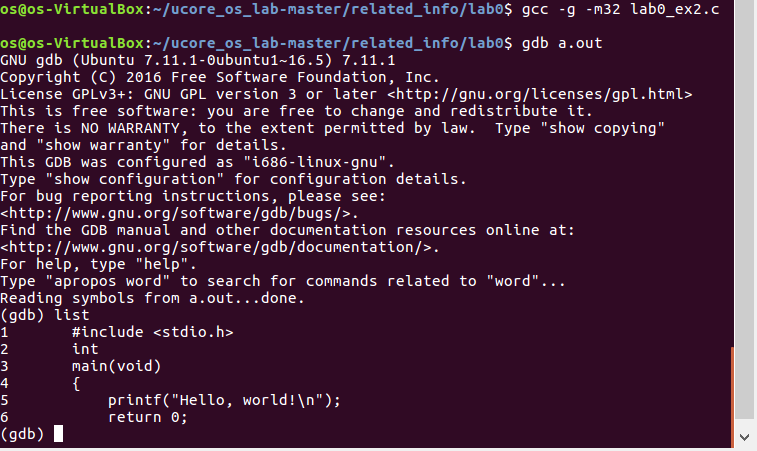
1. 用gdb调试

尝试下面的命令，

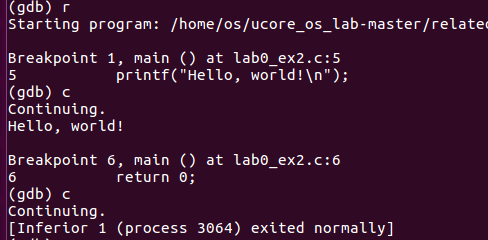
$gcc -g -m32 lab0\_ex2.c

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。

**调试**



设置断点 并逐条运行（没有变量）



1. 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);

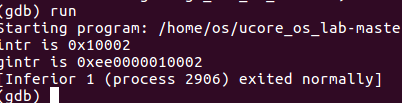
printf("intr is 0x%llx\n", gintr);

return 0;

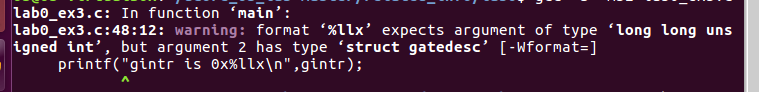
}

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。

**结果**



**错误**



**改进**

**printf("gintr is 0x%llx\n",\*(unsigned long long \*)&(gintr));**

**然后就不会有错误提示了**

**因为是 longlong型的 所以找到gintr地址 转成指向ll型指针 就可以了**

**或者修改 0x%11x 应该也行**

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。

#include <stdio.h>

#include <stdlib.h>

#include "defs.h"

struct list\_entry {

struct list\_entry \*prev, \*next;

};

typedef struct list\_entry list\_entry\_t;

struct entry {

list\_entry\_t node;

int num;

};

int main() {

struct entry head;

list\_entry\_t\* p = &head.node;

list\_init(p);

head.num = 0;

int i;

for (i = 1; i != 26; i ++) {

struct entry \* e = (struct entry \*)malloc(sizeof(struct entry));

e->num = i;

list\_add(p, &(e->node));

p = list\_next(p);

}

//reverse list all node

printf("%d\n",((struct entry \*)p)->num);

while ((p = list\_prev(p)) != &head.node)

printf("%d\n", ((struct entry \*)p)->num);

return 0;

}

#ifndef \_\_LIBS\_LIST\_H\_\_

#define \_\_LIBS\_LIST\_H\_\_

#ifndef \_\_ASSEMBLER\_\_

#include "defs.h"

/\* \*

\* Simple doubly linked list implementation.

\*

\* Some of the internal functions ("\_\_xxx") are useful when manipulating

\* whole lists rather than single entries, as sometimes we already know

\* the next/prev entries and we can generate better code by using them

\* directly rather than using the generic single-entry routines.

\* \*/

struct list\_entry {

struct list\_entry \*prev, \*next;

};

typedef struct list\_entry list\_entry\_t;

static inline void list\_init(list\_entry\_t \*elm) \_\_attribute\_\_((always\_inline));

static inline void list\_add(list\_entry\_t \*listelm, list\_entry\_t \*elm) \_\_attribute\_\_((always\_inline));

static inline void list\_add\_before(list\_entry\_t \*listelm, list\_entry\_t \*elm) \_\_attribute\_\_((always\_inline));

static inline void list\_add\_after(list\_entry\_t \*listelm, list\_entry\_t \*elm) \_\_attribute\_\_((always\_inline));

static inline void list\_del(list\_entry\_t \*listelm) \_\_attribute\_\_((always\_inline));

static inline void list\_del\_init(list\_entry\_t \*listelm) \_\_attribute\_\_((always\_inline));

static inline bool list\_empty(list\_entry\_t \*list) \_\_attribute\_\_((always\_inline));

static inline list\_entry\_t \*list\_next(list\_entry\_t \*listelm) \_\_attribute\_\_((always\_inline));

static inline list\_entry\_t \*list\_prev(list\_entry\_t \*listelm) \_\_attribute\_\_((always\_inline));

static inline void \_\_list\_add(list\_entry\_t \*elm, list\_entry\_t \*prev, list\_entry\_t \*next) \_\_attribute\_\_((always\_inline));

static inline void \_\_list\_del(list\_entry\_t \*prev, list\_entry\_t \*next) \_\_attribute\_\_((always\_inline));

/\* \*

\* list\_init - initialize a new entry

\* @elm: new entry to be initialized

\* \*/

static inline void

list\_init(list\_entry\_t \*elm) {

elm->prev = elm->next = elm;

}

/\* \*

\* list\_add - add a new entry

\* @listelm: list head to add after

\* @elm: new entry to be added

\*

\* Insert the new element @elm \*after\* the element @listelm which

\* is already in the list.

\* \*/

static inline void

list\_add(list\_entry\_t \*listelm, list\_entry\_t \*elm) {

list\_add\_after(listelm, elm);

}

/\* \*

\* list\_add\_before - add a new entry

\* @listelm: list head to add before

\* @elm: new entry to be added

\*

\* Insert the new element @elm \*before\* the element @listelm which

\* is already in the list.

\* \*/

static inline void

list\_add\_before(list\_entry\_t \*listelm, list\_entry\_t \*elm) {

\_\_list\_add(elm, listelm->prev, listelm);

}

/\* \*

\* list\_add\_after - add a new entry

\* @listelm: list head to add after

\* @elm: new entry to be added

\*

\* Insert the new element @elm \*after\* the element @listelm which

\* is already in the list.

\* \*/

static inline void

list\_add\_after(list\_entry\_t \*listelm, list\_entry\_t \*elm) {

\_\_list\_add(elm, listelm, listelm->next);

}

/\* \*

\* list\_del - deletes entry from list

\* @listelm: the element to delete from the list

\*

\* Note: list\_empty() on @listelm does not return true after this, the entry is

\* in an undefined state.

\* \*/

static inline void

list\_del(list\_entry\_t \*listelm) {

\_\_list\_del(listelm->prev, listelm->next);

}

/\* \*

\* list\_del\_init - deletes entry from list and reinitialize it.

\* @listelm: the element to delete from the list.

\*

\* Note: list\_empty() on @listelm returns true after this.

\* \*/

static inline void

list\_del\_init(list\_entry\_t \*listelm) {

list\_del(listelm);

list\_init(listelm);

}

/\* \*

\* list\_empty - tests whether a list is empty

\* @list: the list to test.

\* \*/

static inline bool

list\_empty(list\_entry\_t \*list) {

return list->next == list;

}

/\* \*

\* list\_next - get the next entry

\* @listelm: the list head

\*\*/

static inline list\_entry\_t \*

list\_next(list\_entry\_t \*listelm) {

return listelm->next;

}

/\* \*

\* list\_prev - get the previous entry

\* @listelm: the list head

\*\*/

static inline list\_entry\_t \*

list\_prev(list\_entry\_t \*listelm) {

return listelm->prev;

}

/\* \*

\* Insert a new entry between two known consecutive entries.

\*

\* This is only for internal list manipulation where we know

\* the prev/next entries already!

\* \*/

static inline void

\_\_list\_add(list\_entry\_t \*elm, list\_entry\_t \*prev, list\_entry\_t \*next) {

prev->next = next->prev = elm;

elm->next = next;

elm->prev = prev;

}

/\* \*

\* Delete a list entry by making the prev/next entries point to each other.

\*

\* This is only for internal list manipulation where we know

\* the prev/next entries already!

\* \*/

static inline void

\_\_list\_del(list\_entry\_t \*prev, list\_entry\_t \*next) {

prev->next = next;

next->prev = prev;

}

#endif /\* !\_\_ASSEMBLER\_\_ \*/

#endif /\* !\_\_LIBS\_LIST\_H\_\_ \*/