清华大学操作系统实验ucore-lab0

原创

Love_Star_ 于 2020-03-27 09:54:34 发布 3218 收藏 21

本文链接：https://blog.csdn.net/Love_Star_/article/details/105005607

清华大学操作系统实验ucore-lab0

前面的话

最近开始上网课，大三下学期开了操作系统这门课。然后就想上晚找点资源，发现B站的清华大学这个课还不错，所以最近就在搞这个实验，相关的课堂笔记和实验都会陆续发出来，也不知道能不能做完，今天先发准备实验吧。

先打一针吧，清华大学的实验确实难搞，我也是第一次做。虽然大一学过一段时间Linux，但是搭建环境实验都让我头大，各种百度，各种找方法，了解各种Linux命令，要学习的同学要做好准备，毕竟比较难。

不过万事开头难，中间难，结尾难。不过越难的事情，提升越快。加油吧。

关于Linux

Linux菜鸟教程

我是先把Linux的命令从菜鸟教程里又看了一遍，敲了一遍。不熟悉的同学们也可以这样。

关于环境的搭建

我下载的VMware搭载Ubuntu，然后按照lab0的的要求搭建实验环境，关于Ubuntu的安装，csnd里有许多其他的博客有详细的描述，在这里就不赘述了。

提醒一点，Ubuntu内选择下载源的时候，一定要选择国内的下载源，比如阿里之类的，不然超级慢。我就是这样为了更行gcc，等了一天，然后发现可以用国内的服务器下载。用国内服务器10分钟就解决了。

当然了，你也可以选择他封装好的Ubuntu用Virtualbox打开就行了。

关于如何做实验

uCoreLab实验指导书

uCoreLab实验指导书

uCoreLab实验指导书

重要的事情说三遍，一定要仔仔细细看，不然你有可能环境都搭建不起来。

开始实验

实验一：了解汇编

Try below command

```
$cd related_info/lab0
$gcc -S -m32 lab0_ex1.c
```

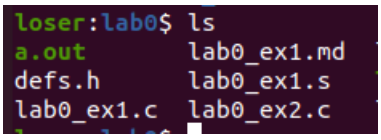
Then you will get lab_ex1.S. Try to understand the content & relation of C file and S file.

运行 gcc -S -m32 lab0_ex1.c，生成S汇编语言文件。

- S表示仅仅编译，不进行链接或汇编
- m32表示生成32位机器的汇编代码

```
int count=1;
int value=1;
int buf[10];//定义不多说
void main()
{
   asm(
 "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
 :
 : "c" (count), "a" (value) , "D" (buf[0])//c：ECX a：EAX D：EDX
 :
      );//这里要注意汇编在Windows和Linux中会汇编的源操作数和目的操作数相反
}
```

按照命令执行之后会得到.s文件



```
1         .file   "lab0_ex1.c"
 2         .text
 3         .globl  count
 4         .data
 5         .align 4
 6         .type   count, @object
 7         .size   count, 4
 8 count:
 9         .long   1
10         .globl  value
11         .align 4
12         .type   value, @object
13         .size   value, 4
14 value:
15         .long   1
16         .comm   buf,40,32
17         .text
18         .globl  main
19         .type   main, @function
20 main:
21 .LFB0:
22         .cfi_startproc
23         endbr32
24         pushl   %ebp
25         .cfi_def_cfa_offset 8
26         .cfi_offset 5, -8
27         movl    %esp, %ebp
28         .cfi_def_cfa_register 5
```

该代码分析的结果就是：把EAX里面的值重复ECX次保存到ES:EDI指向的地址中
就是：buf[0]=value

实验二：用gdb调试

Try below command

```
$cd related_info/lab0
$gcc -g -m32 lab0_ex2.c
```

Then you will get a.out. Try to use gdb to debug lab0_ex2.
运行 gcc -g -m32 lab0_ex2.c得到a.out可执行文件（默认文件名，可以更改名字）
打开实验二的文件：

```
 1 #include <stdio.h>
 2 int main(void)
 3 {
 4     int i=1;
 5     printf("Hello, world!\n");
 6     i=2;
 7     return 0;
 8 }
~
```

用gdb命令调试文件（打开方式有三种，不一一介绍）

```
loser:lab0$ gdb a.out
```

得到如下：

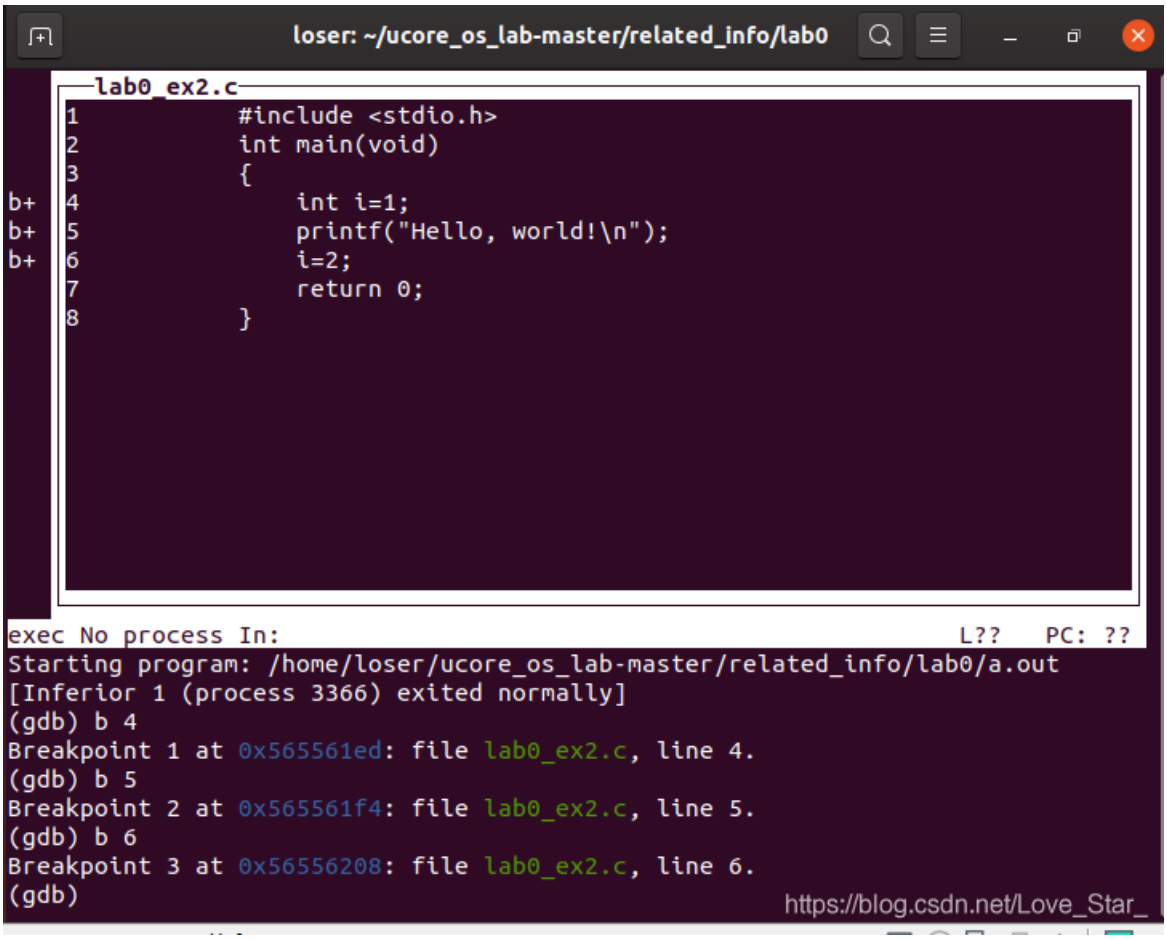
```
loser:lab0$ gdb a.out
GNU gdb (Ubuntu 8.3-0ubuntu1) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...
(gdb)                                                    https://blog.csdn.net/Love_Star_
```

开始调试：

输入如下命令，进如layout查看源码（**虽然用list同样可以，但是list不方便查看源码和汇编代码**）

```
gdb) layout src
```



根据源码设置断点，上图我分别在第4、5、6行设置了断点（上面的b=breakpoints，设置断点，后面的r=run，执行程序，以此类推，不再赘述）

输入info b查看断点信息

```
(gdb) info b
```

```
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x565561ed in main at lab0_ex2.c:4
2       breakpoint     keep y   0x565561f4 in main at lab0_ex2.c:5
3       breakpoint     keep y   0x56556208 in main at lab0_ex2.c:6
4       breakpoint     keep y   <PENDING>  info
(gdb)
```

输入display i 来自动显示变量i的值（i是我为了调试特意加进去的，当然了，你也可以用其他的方式查看变量）
输入命令r之后程序停在第一个断点处，然后依次用n命令往下执行，可以看到i的值在变化。

```
native process 3522 In: main                                  L5    PC: 0x565561f4
(gdb) refresh                                                 6               208
Breakpoint 1, main () at lab0_ex2.c:4
1: i = 1448436305
(gdb) n

Breakpoint 2, main () at lab0_ex2.c:5
1: i = 1
(gdb) n
Hello, world!

Breakpoint 3, main () at lab0_ex2.c:6
1: i = 1
(gdb)
```


简单的调试就完成了，复杂的调试还得慢慢学。

实验三：掌握指针和类型转换相关的 C 编程

给出代码如下：

```
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("gintr is 0x%llx\n",&gintr);

    return 0;
}
```

如果在其他代码段中有如下语句，

```
unsigned intr;
intr=8;
SETGATE(intr, 0,1,2,3);
```

请问执行上述指令后， intr的值是多少？

具体方法：Try below command

```
$cd related_info/lab0
gcc -g -m32 lab0_ex3.c 2>&1|tee make.log
```

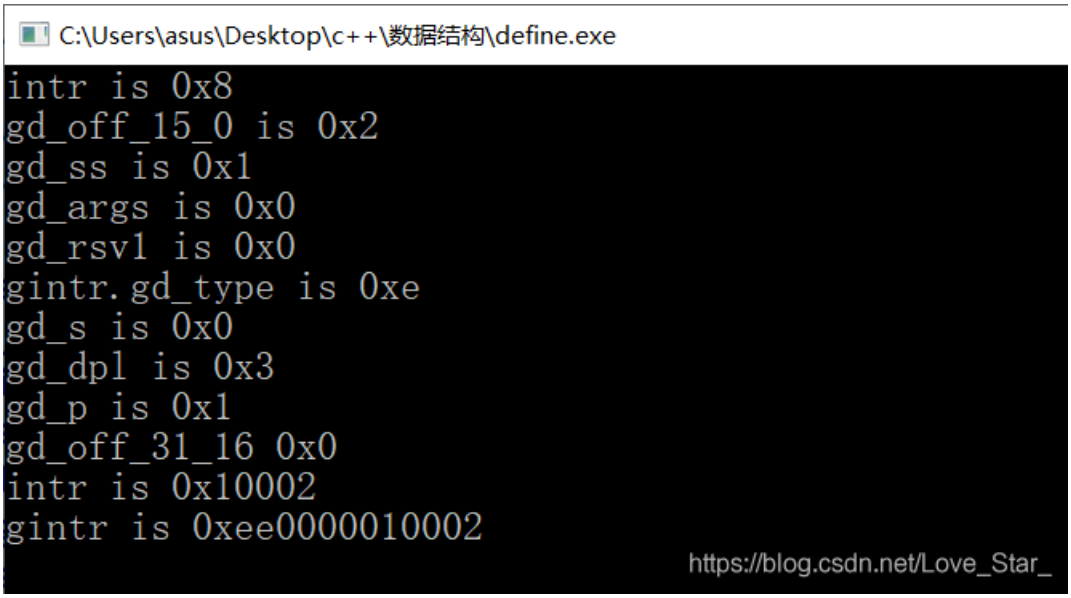
If you get gcc's error, try to read make.log and fix the bugs. If gccsuccessed, then you will get a.out. Try to answer below question.

上面的代码对于我来说，遇到的问题：

- C语言结构体中的冒号用法
- 强制类型转换
- 大小端
- **对于C语言结构体中的冒号用法，可以查看博文1，博文2**
- **对于强制类型转换，可以查看博文**
- **对于大小端输出问题，去百度或者其他博主的文章。**

结构体初始化之后各个位域的值：

函数调用之后的值：



组合一下就得到下图：

gd_off_31_16	gd_p	gd_dpl	gd_s	gd_type	gd_rsv1	gd_args	gd_ss	gd_off_15_0
0000	1	1	1	0	e	0	0	0001	0002

总结：结构体位域先定义的为低位，后定义的为高位。Intel采用小端方式，所以输出为上图所示。

实验四：掌握通用链表结构相关的C编程

用在related_info/lab0/list.h中定义的结构和函数来实现一个小应用程序完成一个基于此链表的数据对象的访问操作。
list.h代码如下：

```
#ifndef __LIBS_LIST_H__
#define __LIBS_LIST_H__

#ifndef __ASSEMBLER__

#include <defs.h>
```

```c
/* *
 * 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_i
nline));
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.
 * */
```

```c
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__ */
```

这写的是一个头文件，我们直接在另外一个文件里调用就好了。然而我用gcc编译的时候出现了很多错误。
一、

```
loser:lab0$ gcc -Wall list.h
list.h:6:10: fatal error: defs.h: 没有那个文件或目录
    6 | #include <defs.h>
      |          ^~~~~~~
compilation terminated.
```

vim 打开这个文件,把 ,<defs.h>改成"defs.h"就好了。
二、

```
loser:lab0$ gcc -Wall list.h
list.h:29:15: error: unknown type name ‘bool’
   29 | static inline bool list_empty(list_entry_t *list) __attribute__((al
ways_inline));
      |               ^~~~
list.h:112:15: error: unknown type name ‘bool’
  112 | static inline bool
      |               ^~~~
```

百度了之后网上大多数解决办法是：在C语言标准(C89)没有定义布尔类型，所以会报错。而C99提供了一个头文件<stdbool.h>定义了bool，true代表1，false代表0。只要导入stdbool.h，就能非常方便的操作布尔类型了。

但是在这里不是这样的，打开defs.h就会发现有这样一句tepydef bool int,只是换了名而已。

正确的解决办法就是：gcc -Wall -m32 -g list.h

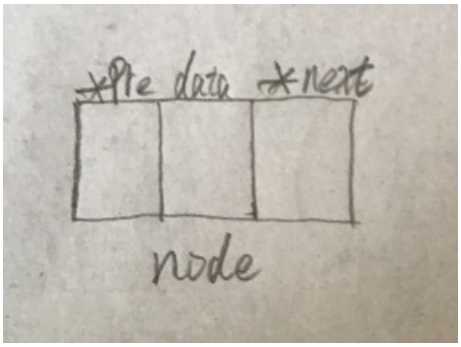
在这里m32指定编译为32位应用程序，其中可能和64位语法有些不同，所以加了就好了，要注意这个问题。

三、

接下来就是链表了，看了list.h文件之后，发现这个双向循环链表和我们以往定义的链表不是一样的结构。

正常定义图如下：

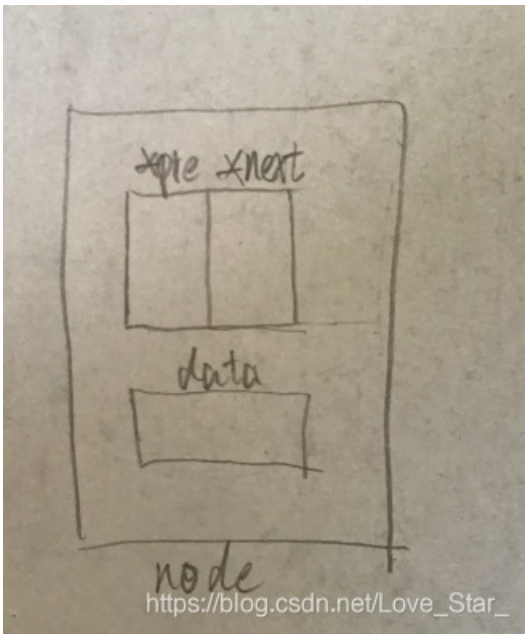
```
typedef struct linklist{
     struct list_entry *prev, *next;
      int v;
  }link;
```



lab0的链表结构：

```
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct linklist{
       list_entry_t node;
       int v;
   }link;
```



也就是说平时我们定义的双向链表链表的指针域和数据域是一个整体（*pre *next,指向整个地址空间），而lab0的指针域和数据域不是一整体，而是平行的关系（*pre *next 指向自己）。

下面列出我写的代码：

```c
#include<stdio.h>
#include <stdlib.h>
#include"list.h"
typedef struct linklist{
    list_entry_t node;
    int v;
}link;
void init(link *linknode){//初始化链表
    linknode->v=0;
   // list_entry_t* p=(linknode->node);
    list_init(&(linknode->node)) ;
}
void add_after(link *oldnode,link *newnode){//某个节点之
    list_add_after(&(oldnode->node),&(newnode->node));
}
void add_before(link *oldnode,link *newnode){//某个节点>
    list_add_before(&(oldnode->node),&(newnode->node));
}
void list_delete(link *linknode){//删除某个节点
list_add_before(&(oldnode->node),&(newnode->node));
}
void list_delete(link *linknode){//删除某个节点
    list_del(&(linknode->node));
}
void create_linklist(link *linknode){//建立链表
    int flag=-1;//设置标志位
    int x;
    list_entry_t *p;
    p=&(linknode->node);
    scanf("%d",&x);
    while(x!=flag){
        link *temp=(link*)malloc(sizeof(link));
        temp->v=x;
        add_after((struct linklist*)p,temp);
        p=p->next;
        scanf("%d",&x);
    }
    }
void printf_linklist(link *linknode){
    list_entry_t *temp;
    temp=&(linknode->node);
    temp=temp->next;
    int count=1;
    while(temp!=&(linknode->node)){
        printf("第%d个元素是：%d\n",count,((struct linkl
        count++;
        temp=temp->next;
    }
}
int main(){
    link* H=(link*)malloc(sizeof(link));
    init(H);
    create_linklist(H);
    printf_linklist(H);
}
```

提示：由于lab0链表结构，所以后边我们要输出定义的数据域必须强制类型转换，这就延伸出一个问题。
正常定义（数据域在指针域后面）

```
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct linklist{
       list_entry_t node;
       int v;
   }link;
```

不正常定义（地址域在指针域前面）

```
struct list_entry {
    struct list_entry *prev, *next;
};

typedef struct linklist{
  int v;
       list_entry_t node;
  }link;
```

正常定义和非正常定义导致强制类型转换后所对应的内存空间是不一样的，非正常定义会导致强制类型转换的数据域不是你定义的数据域。这里涉及到强制类型装换的问题，有兴趣朋友可以查看其他博主的博文。

分隔符

终于做完了lab0，感觉路还很远，知道的愈多，知道的愈少，加油吧，少年。。。