中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

操作系统内核分析与实践

2. Linux内核源代码导读

授课教师：游伟 副教授

授课时间：周五16:00 – 17:30（公教三楼3505）

上机时间：周五18:00 – 19:30（理工配楼208B机房）

课程主页：https://www.youwei.site/course/kernel

目录

2.1 内核关键数据结构

- 链表
- 映射
- 缓冲队列
- 位数组

……

更多内容：《Linux内核设计与实现（第3版）》第6章

2.1.1 链表

- 循环双向链表

  (Simple doubly linked list)

- 哈希链表

  (Doubly linked list with a single pointer list head)

- 无锁单链表

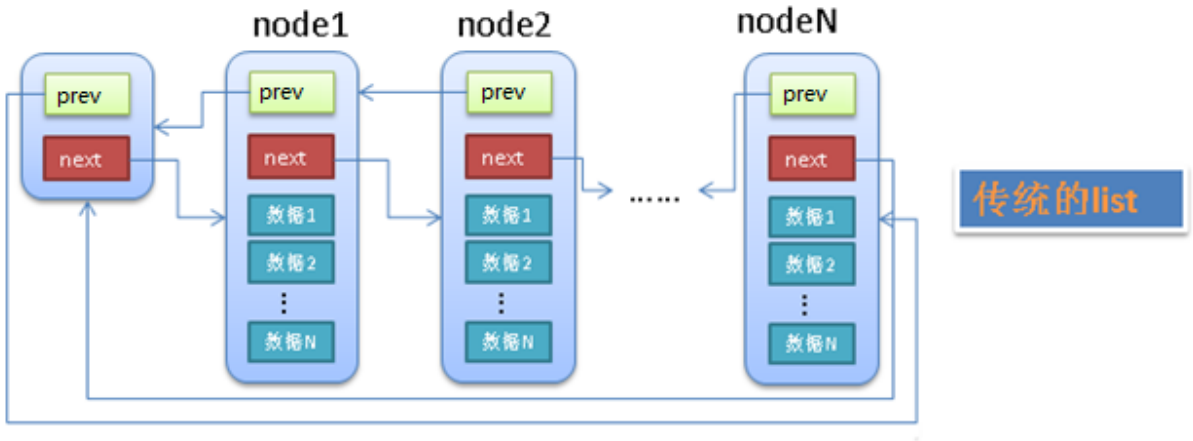
  (Lock-less NULL terminated single linked list)

- 降序优先排序的双向链表

  (Descending-priority-sorted double-linked list)

* 循环双向链表

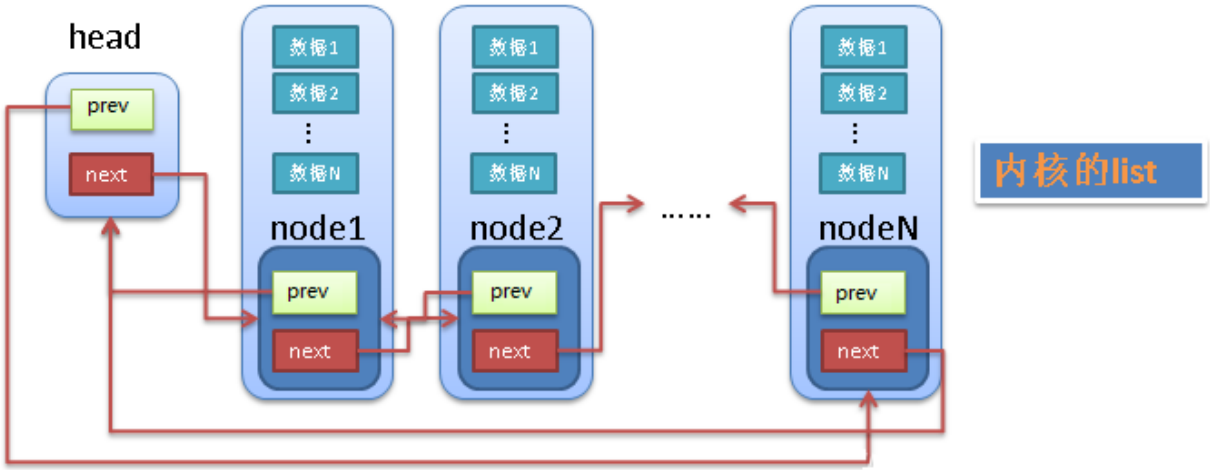
- 传统的循环双向链表
  - 有个单独的头节点（head）
  - 每个节点（node）除包含必要的数据之外，还有2个指针：pre指针指向前一个节点，next指针指向后一个节点
  - 头结点的pre指针指向链表的最后一个节点
  - 链表的最后一个节点的next指针指向头结点




```
1.   typedef struct node
2.   {
3.     int data;
4.     /* … */
5.     struct node *prev;
6.     struct node *next;
7.   } Node;
```

* 循环双向链表

- Linux内核中的循环双向链表
  - Linux内核中的链表不是将用户数据保存在链表节点中，而是将链表节点保存在用户数据中
  - Linux内核中的链表节点只有2个指针（pre和next），链表的节点将独立于用户数据之外，便于实现链表的共同操作




```
1.   struct list_head
2.   {
3.     struct list_head *prev;
4.     struct list_head *next;
5.   };

6.   typedef struct node
7.   {
8.     int data;
9.     /* … */
10.    struct list_head list;
11.  } Node;
```

实例

```
// File: chapter2/VRUC/student.c(.h)

1.   #include <linux/list.h>
2.   #include <linux/slab.h>

3.   struct student
4.   {
5.     int id;
6.     char name[16];
7.     struct list_head list;
8.   };

9.   extern struct list_head student_list;

10.  void add_student_to_list(struct student *student);
11.  void del_student_from_list(struct student *student);
12.  struct student * create_student(int id, char *name);
13.  void show_student(struct list_head *node);
14.  void list_students(void);

15.  LIST_HEAD(student_list);

16.  void add_student_to_list(struct student *student)
17.  {
18.    list_add(&student->list, &student_list);
19.  }

20.  void del_student_from_list(struct student *student)
21.  {
22.    list_del(&student->list);
23.  }
```

```
24.  struct student * create_student(int id, char *name) {
25.    struct student *stu;

26.    stu = kmalloc(sizeof(struct student), GFP_KERNEL);
27.    stu->id = id;
28.    strcpy(stu->name, name);
29.    INIT_LIST_HEAD(&stu->list);

30.    return stu;
31.  }

32.  void show_student(struct list_head *node) {
33.    struct student *stu;
34.    stu = list_entry(node, struct student, list);
35.    printk("student: %d %s\n", stu->id, stu->name);
36.  }

37.  void list_students(void) {
38.    struct list_head *node;
39.    list_for_each(node, &student_list) {
40.      show_student(node);
41.    }
42.  }

43.  void search_studenet(const char *name)
44.  {
45.    struct student *stu;
46.    list_for_each_entry(stu, &student_list, list) {
47.      if (my_strcmp(stu->name, name) == 0)
48.        show_student(&stu->list);
49.    }
50.  }
```

定义与初始化

```
1  static inline void INIT_LIST_HEAD(struct list_head *list)
2  {
3      WRITE_ONCE(list->next, list);
4      list->prev = list;
5  }
6
7  #define LIST_HEAD_INIT(name) { &(name), &(name) }
8
9  #define LIST_HEAD(name) \
10     struct list_head name = LIST_HEAD_INIT(name)
```

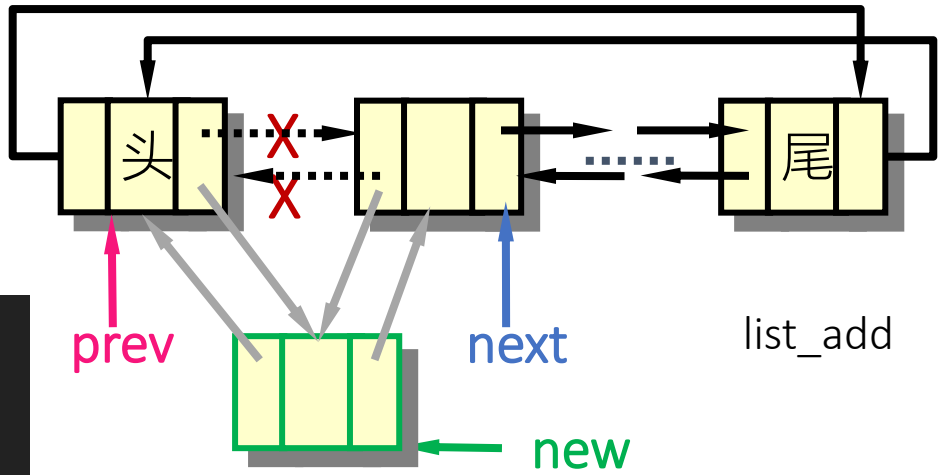
← 动态初始化

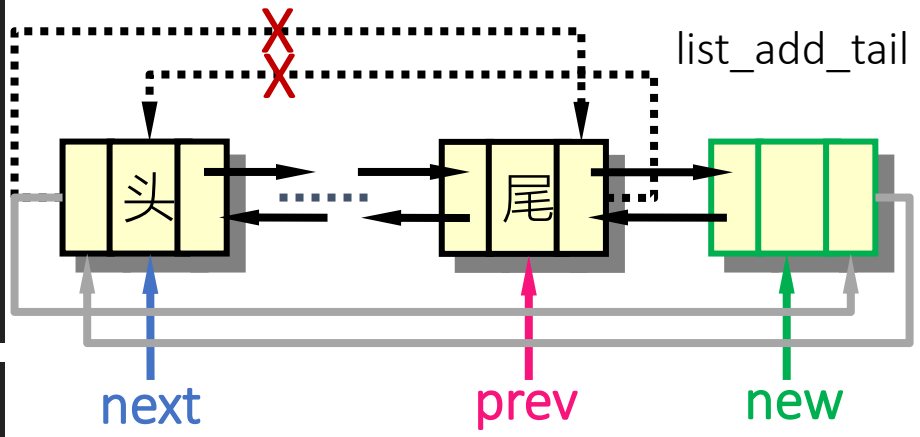
← 静态初始化

```
13 #define offsetof(TYPE, MEMBER)  ((size_t)&((TYPE *)0)->MEMBER)
14
15 #define container_of(ptr, type, member) ({                  \
16     void *__mptr = (void *)(ptr);                    \
17     BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&   \
18                      !__same_type(*(ptr), void),             \
19                      "pointer type mismatch in container_of()");    \
20     ((type *)(__mptr - offsetof(type, member))); })
21
22 #define list_entry(ptr, type, member) \
23     container_of(ptr, type, member)
```

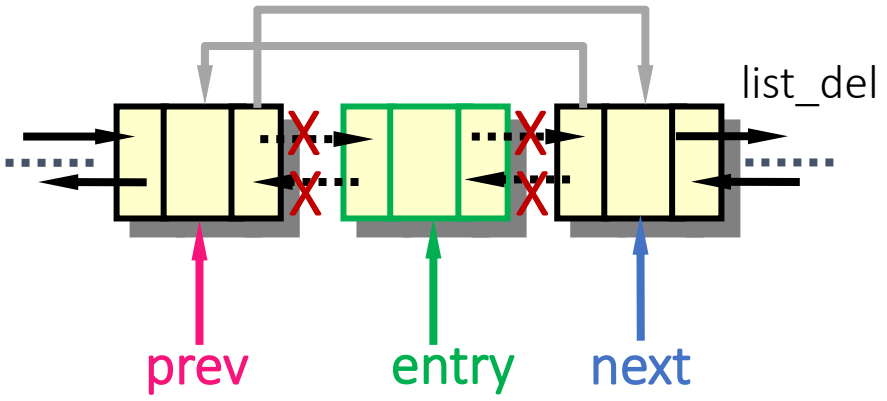
添加和删除

```c
static inline void __list_add(struct list_head *new,
                              struct list_head *prev,
                              struct list_head *next)
{
    if (!__list_add_valid(new, prev, next))
        return;

    next->prev = new;
    new->next = next;
    new->prev = prev;
    WRITE_ONCE(prev->next, new);
}

static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}

static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
    __list_add(new, head->prev, head);
}
```






```c
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
    next->prev = prev;
    WRITE_ONCE(prev->next, next);
}

static inline void __list_del_entry(struct list_head *entry)
{
    if (!__list_del_entry_valid(entry))
        return;

    __list_del(entry->prev, entry->next);
}

static inline void list_del(struct list_head *entry)
{
    __list_del_entry(entry);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}
```



遍历

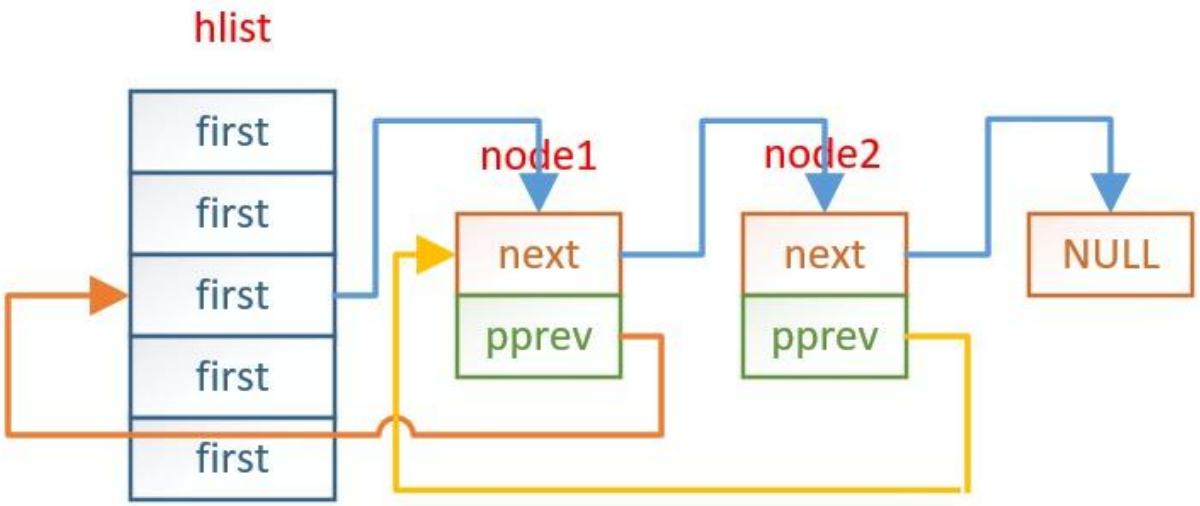
```c
// list_first_entry: get the first element from a list
#define list_first_entry(ptr, type, member) list_entry((ptr)->next, type, member)

// list_last_entry: get the last element from a list
#define list_last_entry(ptr, type, member) list_entry((ptr)->prev, type, member)

// list_next_entry: get the next element in list
#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)

// list_prev_entry: get the prev element in list
#define list_prev_entry(pos, member) \
    list_entry((pos)->member.prev, typeof(*(pos)), member)

// list_for_each: iterate over a list
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

// list_for_each_safe: iterate over a list safe against removal of list entry
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); pos = n, n = pos->next)

// list_for_each_prev: iterate over a list backwards
#define list_for_each_prev(pos, head) \
    for (pos = (head)->prev; pos != (head); pos = pos->prev)

// list_for_each_prev_safe: iterate over a list backwards safe against removal of list entry
#define list_for_each_prev_safe(pos, n, head) \
    for (pos = (head)->prev, n = pos->prev; pos != (head); pos = n, n = pos->prev)
```

遍历

```
1  // list_for_each_entry: iterate over list of given type
2  #define list_for_each_entry(pos, head, member)                \
3      for (pos = list_first_entry(head, typeof(*pos), member);    \
4           &pos->member != (head);                                \
5           pos = list_next_entry(pos, member))
6
7  // list_for_each_entry_safe: iterate over list of given type safe against removal of list entry
8  #define list_for_each_entry_safe(pos, n, head, member)          \
9      for (pos = list_first_entry(head, typeof(*pos), member),    \
10         n = list_next_entry(pos, member);                       \
11          &pos->member != (head);                                \
12          pos = n, n = list_next_entry(n, member))
13
14 // list_for_each_entry_reverse: iterate backwards over list of given type
15 #define list_for_each_entry_reverse(pos, head, member)          \
16     for (pos = list_last_entry(head, typeof(*pos), member);     \
17          &pos->member != (head);                                \
18          pos = list_prev_entry(pos, member))
19
20 // list_for_each_entry_safe_reverse - iterate backwards over list safe against removal
21 #define list_for_each_entry_safe_reverse(pos, n, head, member)      \
22     for (pos = list_last_entry(head, typeof(*pos), member),     \
23         n = list_prev_entry(pos, member);                       \
24          &pos->member != (head);                                \
25          pos = n, n = list_prev_entry(n, member))
```

* 哈希链表

- 哈希链表：在对需要存储的数据进行hash时，如果产生了冲突，就使用链表的方式将产生冲突的数据进行存储
- 哈希链表是循环双向链表的变体
  - 双向非循环链表
  - 表头和结点有不同的结构体定义




```
1.   struct hlist_head
2.   {
3.     struct hlist_node *first;
4.   };

5.   struct hlist_node
6.   {
7.     struct hlist_node *next;
8.     struct hlist_node **pprev;
9.   };
```

实例

```
// File: chapter2/VRUC/student_hash.c(.h)

#include <linux/slab.h>
#include <linux/hashtable.h>

struct student {
    int id;
    char name[16];
    struct hlist_node hnode;
};

#define BITS 8
#define BUCKETS (1 << BITS)

static struct hlist_head student_table[BUCKETS];

static inline unsigned int hashfn(int id)
{
    return (unsigned int)id & (BUCKETS - 1);
}

static void add_student(struct student *stu)
{
    unsigned int b = hashfn(stu->id);
    hlist_add_head(&stu->hnode, &student_table[b]);
}

static void del_student(struct student *stu)
{
    hlist_del(&stu->hnode);
    kfree(stu);
}

static void list_bucket(int key)
{
    unsigned int b = hashfn(key);
    struct student *stu;
    hlist_for_each_entry(stu, &student_table[b], hnode) {
        printk("student: %d %s\n", stu->id, stu->name);
    }
}

static struct student *find_student(int id)
{
    unsigned int b = hashfn(id);
    struct student *stu;
    hlist_for_each_entry(stu, &student_table[b], hnode) {
        if (stu->id == id)
            return stu;
    }
    return NULL;
}

struct student * create_student(int id, char *name) {
  struct student *stu;

  stu = kmalloc(sizeof(struct student), GFP_KERNEL);
  stu->id = id;
  strcpy(stu->name, name);
  INIT_HLIST_HEAD(&stu->node);

  return stu;
}
```

定义与初始化

```
1  #define HLIST_HEAD_INIT { .first = NULL }
2  #define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
3  #define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
4
5  static inline void INIT_HLIST_NODE(struct hlist_node *h)
6  {
7      h->next = NULL;
8      h->pprev = NULL;
9  }
10
11 static inline int hlist_unhashed(const struct hlist_node *h)
12 {
13     return !h->pprev;
14 }
15
16 static inline int hlist_empty(const struct hlist_head *h)
17 {
18     return !READ_ONCE(h->first);
19 }
```

添加

```c
static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first) first->pprev = &n->next;
    WRITE_ONCE(h->first, n);
    n->pprev = &h->first;
}

static inline void hlist_add_before(struct hlist_node *n, struct hlist_node *next)
{
    n->pprev = next->pprev;
    n->next = next;
    next->pprev = &n->next;
    WRITE_ONCE(*(n->pprev), n);
}

static inline void hlist_add_behind(struct hlist_node *n, struct hlist_node *prev)
{
    n->next = prev->next;
    WRITE_ONCE(prev->next, n);
    n->pprev = &prev->next;
    if (n->next) n->next->pprev  = &n->next;
}

static inline void hlist_add_fake(struct hlist_node *n)
{
    n->pprev = &n->next;
}
```

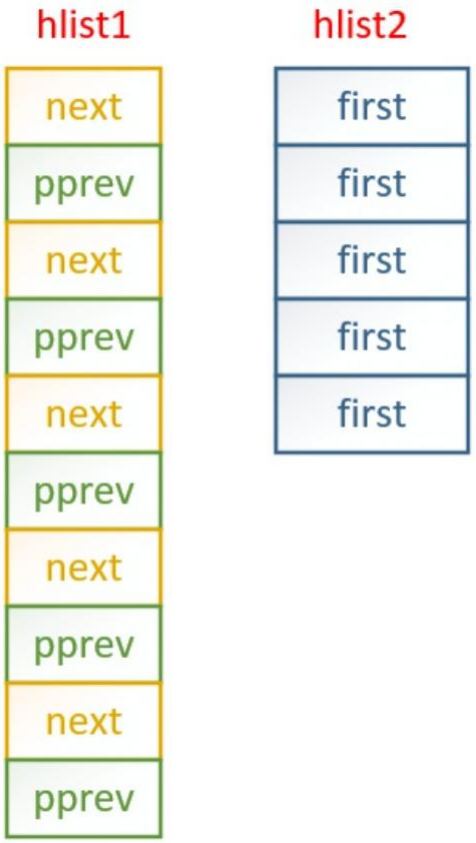
删除

```
1  static inline void __hlist_del(struct hlist_node *n)
2  {
3      struct hlist_node *next = n->next;
4      struct hlist_node **pprev = n->pprev;
5
6      WRITE_ONCE(*pprev, next);
7      if (next)
8          next->pprev = pprev;
9  }
10
11 static inline void hlist_del(struct hlist_node *n)
12 {
13     __hlist_del(n);
14     n->next = LIST_POISON1;
15     n->pprev = LIST_POISON2;
16 }
17
18 static inline void hlist_del_init(struct hlist_node *n)
19 {
20     if (!hlist_unhashed(n)) {
21         __hlist_del(n);
22         INIT_HLIST_NODE(n);
23     }
24 }
```

遍历

```
#define hlist_entry(ptr, type, member) container_of(ptr,type,member)

#define hlist_for_each(pos, head) \
    for (pos = (head)->first; pos ; pos = pos->next)

#define hlist_for_each_safe(pos, n, head) \
    for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
         pos = n)

#define hlist_entry_safe(ptr, type, member) \
    ({ typeof(ptr) ____ptr = (ptr); \
       ____ptr ? hlist_entry(____ptr, type, member) : NULL; \
    })

// hlist_for_each_entry: iterate over list of given type
#define hlist_for_each_entry(pos, head, member)              \
    for (pos = hlist_entry_safe((head)->first, typeof(*(pos)), member);\
         pos;                                \
         pos = hlist_entry_safe((pos)->member.next, typeof(*(pos)), member))

// hlist_for_each_entry_safe: iterate over list of given type safe against removal of list entry
#define hlist_for_each_entry_safe(pos, n, head, member)          \
    for (pos = hlist_entry_safe((head)->first, typeof(*pos), member);\
         pos && ({ n = pos->member.next; 1; });          \
         pos = hlist_entry_safe(n, typeof(*pos), member))
```

思考：表头结构为何与普通结点结构不同?



目的是节省空间。 hash表中同样包含5个元素，hlist2占用的空间为hlist1的1/2。

思考：为什么是**prev，而不是*prev?

```
1.  struct hlist_head
2.  {
3.    struct hlist_node *first;
4.  };

5.  struct hlist_node
6.  {
7.    struct hlist_node *next, *prev;
8.  };

9.  static void hlist_add(struct hlist_head *head,
10.                       struct hlist_node *new_node)
11. {
12.   if(NULL != head->first)
13.   {
14.     new_node->next = head->first->next;
15.     head->first->prev = new_node;
16.   }
17.   else
18.   {
19.     new_node->next = NULL;
20.   }
21.   new_node->prev = NULL;
22.   head->first = new_node;
23. }
```

```
24. static void hlist_del(struct hlist_head *head,
25.                        struct hlist_node *del_node)
26. {
27.   if(head->first == del_node)
28.   {
29.     head->first = del_node->next;
30.     if(NULL != del_node->next)
31.     {
32.       del_node->next->prev = NULL;
33.     }
34.   }
35.   else
36.   {
37.     del_node->prev->next = del_node->next;
38.     if(NULL != del_node->next)
39.     {
40.       del_node->next->prev = del_node->prev;
41.     }
42.   }
43. }
```

目的是提高效率。 在删除结点时，普通双向链表需要单独判断是否是head结点，而使用二级指针则无需额外判断。

2.2 内联汇编

- 虽然Linux的核心代码大部分是用C语言编写的，但是不可避免的其中还是有一部分是用汇编语言写成的
- 使用汇编语言的几点原因：
  - 提高效率，例如使用内联汇编实现一个高效的strcmp
  - 直接和硬件打交道需要用到的指令，C语言没有对应的语句
  - CPU中一些特殊指令和新增指令，没有对应的C语句
- Linux内核的汇编语言代码有两种形式：
  - 一种是直接写成汇编语言源程序的形式，主要是一些Linux的启动代码
  - 另一种是利用GCC的内联汇编语言语句asm嵌在Linux的C语言代码中的

更多内容：https://blog.csdn.net/notbaron/category_10007974.html

Intel v.s. AT&T

Intel Code	AT&T Code
mov eax,1	movl $1,%eax
mov ebx,0ffh	movl $0xff,%ebx
int 80h	int $0x80
mov ebx, eax	movl %eax, %ebx
mov eax,[ecx]	movl (%ecx),%eax
mov eax,[ebx+3]	movl 3(%ebx),%eax
mov eax,[ebx+20h]	movl 0x20(%ebx),%eax
add eax,[ebx+ecx*2h]	addl (%ebx,%ecx,0x2),%eax
lea eax,[ebx+ecx]	leal (%ebx,%ecx),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

2.2.1 AT&T汇编语法

- 源操作数与目的操作数顺序
  - 第一操作数为源操作数，第二操作数为目的操作数
  - 例： mov %eax, %ebx // 将eax寄存器的数据移至ebx寄存器
- 寄存器命名
  - 寄存器使用 "%" 前缀标识
  - 例：%eax //eax寄存器
- 立即数
  - 立即数以 "$" 为前缀，十六进制常量用 "0x" 进行标识
  - 例：mov $0x10,%eax //将立即数0x10赋值给寄存器eax

2.2.1 AT&T汇编语法

- 操作数长度大小
  - 存储器操作数的大小取决于操作码名字的最后一个字符
  - 操作码后缀 'b' 、'w'、'l'、'q' 分别指明了字节（BYTE，8位）、字（WORD，16位）、双字（long，32位）、四字（QUADWORD，64位）存储器引用
  - 如果省略后缀，将通过使用的寄存器大小来猜测指令的操作数长度
  - 例：mov %eax, %ebx // movl %eax, %ebx，操作数长度32位
    mov (%eax), %bh // movb (%eax), %bh，操作数长度8位
    movw $1, (%eax) //操作数长度16位
- 寄存器寻址
  - 基本格式：segment:disp(base, index, scale)
  - 寻址方案：segment:[base + index * scale + disp]
  - 例：subl -0x20(%ebx, %ecx, 0x4), %eax //eax ← eax – [ebx+ecx*4h-20h]

示例：在屏幕上打印 Hello World

```
1.  # AT&T 格式
2.  # Hello.s

3.  .data #数据段声明
4.      msg : .string "Hello World!\\n"        # 要输出的字符串
5.      len = . - msg                      # 字符串长度
6.  .text                   # 代码段声明
7.  .global _start          # 指定入口函数

8.  _start:
9.                          # 在屏幕上显示一个字符串
10.     movl $len, %edx     # 参数三:字符串长度
11.     movl $msg, %ecx     # 参数二:要显示的字符串
12.     movl $1, %ebx       # 参数一:文件描述符 (stdout)
13.     movl $4, %eax       # 系统调用号 (sys_write)
14.     int $0x80           # 调用内核功能
15.                         # 退出程序
16.     movl $0, %ebx       # 参数一:退出代码
17.     movl $1, %eax       # 系统调用号 (sys_exit)
18.     int $0x80           # 调用内核功能
```

示例：循环累加

```
// C语言代码: gcc -g -o test test.c

1.   #include <stdio.h>

2.   int main()
3.   {
4.     int i, sum = 0;

5.     for (int i = 1; i <= 10; i++) sum += i;

6.     printf("sum: %d\n", sum);
7.   }

// 汇编代码: objdump -d -S test > test.dump

539:   c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)         // sum ← 0
540:   c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%ebp)          // i ← 1
547:   eb 0a                   jmp    553 <main+0x36>
549:   8b 55 f4                mov    -0xc(%ebp),%edx          // edx ← i
54c:   01 55 f0                add    %edx,-0x10(%ebp)         // sum ← sum + i
54f:   83 45 f4 01             addl   $0x1,-0xc(%ebp)          // i++
553:   83 7d f4 0a             cmpl   $0xa,-0xc(%ebp)          // i <= 10?
557:   7e f0                   jle    549 <main+0x2c>
559:   83 ec 08                sub    $0x8,%esp
55c:   ff 75 f0                pushl  -0x10(%ebp)              // sum
55f:   8d 90 38 e6 ff ff       lea    -0x19c8(%eax),%edx
565:   52                      push   %edx                     // "sum: %d\n"
566:   89 c3                   mov    %eax,%ebx
568:   e8 43 fe ff ff          call   3b0 <printf@plt>         // call printf
```

2.2.2 基础内联汇编

- 基本说明 `asm volatile ("assembly code");`
  - 使用asm关键字指出汇编语言编写的代码段落
  - 如果volatile修饰符，则表示不希望编译器优化该内联汇编代码段
  - 语句之间使用 ";"、"\n" 、"\n\t" 分开
  - 基础内联汇编代码通过C语言全局变量名称的方式整合输入值和输出值
- 代码示例：计算 c = a + b

```
// should be compiled with -m32 -fno-pic

1.   #include <stdio.h>
2.   int a, b, c;

3.   int main()
4.   {
5.     a = 1, b = 2;
6.     asm volatile (
7.       "movl a, %eax\n\t"
8.       "addl b, %eax\n\t"
9.       "movl %eax, c\n\t"
10.    );
11.    printf("c: %d\n", c);
12.  }
```

```
56e:   c7 05 14 20 00 00 01    movl   $0x1,0x2014          // a ← 1
575:   00 00 00
578:   c7 05 0c 20 00 00 02    movl   $0x2,0x200c          // b ← 2
57f:   00 00 00
582:   a1 14 20 00 00          mov    0x2014,%eax          // eax ← a
587:   03 05 0c 20 00 00       add    0x200c,%eax          // eax ← eax + b
58d:   a3 10 20 00 00          mov    %eax,0x2010          // c ← eax
592:   a1 10 20 00 00          mov    0x2010,%eax
597:   83 ec 08                sub    $0x8,%esp
59a:   50                      push   %eax
59b:   68 40 06 00 00          push   $0x640
5a0:   e8 fc ff ff ff          call   5a1 <main+0x44>
```

2.2.3 扩展内联汇编

- 扩展内联汇编的基本格式

```
asm volatile ("assembly code"
              : output operands                  /* optional */
              : input operands                   /* optional */
              : list of clobbered registers /* optional */
              );
```

- 代码示例：计算 c = a + b

```
// should be compiled with -m32 -fno-pic

1.   int main()
2.   {
3.     int a = 1, b = 2, c = 0;

4.     asm volatile
5.     (
6.       "addl %2, %0\n\t"
7.       : "=g"(c)
8.       : "0"(a), "g"(b)
9.       : "memory"
10.    );

11.    printf("c: %d\n", c);
12.  }
```

```
53a:   c7 45 ec 01 00 00 00    movl   $0x1,-0x14(%ebp)      // a = 1
541:   c7 45 f0 02 00 00 00    movl   $0x2,-0x10(%ebp)      // b = 2
548:   c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)       // c = 0
54f:   8b 45 ec                mov    -0x14(%ebp),%eax      // eax ← a
552:   03 45 f0                add    -0x10(%ebp),%eax      // eax ← eax + b
555:   89 45 f4                mov    %eax,-0xc(%ebp)       // c ←eax
558:   83 ec 08                sub    $0x8,%esp
55b:   ff 75 f4                pushl  -0xc(%ebp)
55e:   8d 82 28 e6 ff ff       lea    -0x19d8(%edx),%eax
564:   50                      push   %eax
565:   89 d3                   mov    %edx,%ebx
567:   e8 44 fe ff ff          call   3b0 <printf@plt>
```

2.2.3 扩展内联汇编

- Assembly Code：
  - 在汇编中用“%数字”来代表输出/输入操作数，数字从0开始编号，依次表示输出部分和输入部分从左到右的参数
  - 为了与操作数区分开来，寄存器用两个%引出，如：%%eax
  - 使用标签时有两个限制：只能跳转到相同asm段内的标签，不能从一个asm段跳转到另一个asm段中的标签；汇编后的代码清单不能使用相同的标签
  - 数字标签：条件分支和无条件分支都允许指定一个数字加上方向标志（f/b）作为标签，方向标志指出处理器应该向哪个方向查找数字型标签

2.2.3 扩展内联汇编

- Output/Input Operands：
  - 输入操作数标准形式： "constraints" (C expression)
  - 输出操作数标准形式： "constraints modifiers constraints" (C expression)
  - 约束（constraints）
    - 寄存器操作数约束
    - 内存操作数约束
    - 匹配约束

`a,b,c,d,S,D` 分别代表 `eax,ebx,ecx,edx,esi,edi` 寄存器
`r` 上面的寄存器的任意一个（谁闲着就用谁）
`m` 内存
`i` 立即数（常量，只用于输入操作数）
`g` 使用任何可用的寄存器或者内存位置

  - 修改标记（constraint modifiers），只用于output operands
    - +：表示被修饰的操作数可读可写
    - =：表示被修饰的操作数只能写入
    - &：被其修饰的Output操作表达式要在所有的Input操作表达式被输入之前输出

（先使用输出值对被&修饰的Output操作表达式进行填充，然后对Input操作表达式进行输入）

2.2.3 扩展内联汇编

- List of Clobbered Registers：
  - 编译器假设输入值和输出值使用的寄存器会被改动，并且相应做出处理，不需要在Clobbered Registers中包含这些值
  - 如果内联汇编代码使用了没有初始化地声明为输入值或者输出值的任何其它寄存器，则要通知编译器，以便避免使用它们
  - 如果在内联汇编代码中使用了没有在输入值或输出值中定义的任何内存位置，那它必须被标记为被破坏的。在改动部分中使用“memory”通知编译器这个内存位置在内联汇编代码中被改动

例子：比较两个数的大小

```
// 使用普通标签

1.   #include <stdio.h>
2.
3.   int main()
4.   {
5.       int a = 10;
6.       int b = 20;
7.       int result;
8.
9.       asm("cmp %1, %2\n\t"
10.          "jge greater\n\t"
11.          "movl %1, %0\n\t"
12.          "jmp end\n"
13.          "greater:\n\t"
14.          "movl %2, %0\n"
15.          "end:"
16.          :"=r"(result)
17.          :"r"(a), "r"(b));
18.
19.      printf("The larger value is %d\n", result);
20.      return 0;
21.  }
```

```
// 使用数字标签

1.   #include <stdio.h>
2.
3.   int main()
4.   {
5.       int a = 10;
6.       int b = 20;
7.       int result;
8.
9.       asm("cmp %1, %2\n\t"
10.          "jge 0f\n\t"
11.          "movl %1, %0\n\t"
12.          "jmp 1f\n"
13.          "0:\n\t"
14.          "movl %2, %0\n"
15.          "1:"
16.          :"=r"(result)
17.          :"r"(a), "r"(b));
18.
19.      printf("The larger value is %d\n", result);
20.      return 0;
21.  }
```

例子：内核代码中strcpy的实现

```
1.   char *strcpy(char *dest, const char *src)
2.   {
3.     int d0, d1, d2;
4.
5.     asm volatile
6.     (
7.       "1:\tlodsb\n\t"
8.       "stosb\n\t"
9.       "testb %%al,%%al\n\t"
10.      "jne 1b"
11.      : "=&S" (d0), "=&D" (d1), "=&a" (d2)
12.      : "0" (src), "1" (dest) : "memory"
13.    );
14.
15.    return dest;
16.  }

17.  EXPORT_SYMBOL(strcpy);
```

```
51f: 8b 55 0c           mov 0xc(%ebp),%edx
522: 8b 45 08           mov 0x8(%ebp),%eax
525: 89 d1              mov %edx,%ecx
527: 89 c2              mov %eax,%edx
529: 89 ce              mov %ecx,%esi
52b: 89 d7              mov %edx,%edi
52d: ac                 lods %ds:(%esi),%al
52e: aa                 stos %al,%es:(%edi)
52f: 84 c0              test %al,%al
531: 75 fa              jne 52d <strcpy+0x20>
533: 89 fa              mov %edi,%edx
535: 89 f1              mov %esi,%ecx
537: 89 4d ec           mov %ecx,-0x14(%ebp)
53a: 89 55 f0           mov %edx,-0x10(%ebp)
53d: 89 45 f4           mov %eax,-0xc(%ebp)
540: 8b 45 08           mov 0x8(%ebp),%eax
543: 83 c4 10           add $0x10,%esp
```

```
;使用格式，指令包括 STOSB，STOSW
STOSB     ;字节串存储：ES:[DI]<-AL
          ;DI<-DI+1
```

```
;使用格式，指令包括LODSB，LODSW
LODSB     ;字节串存储：AL<-DS:[SI]
          ;SI<-SI+-1
```

例子： Clobbered Register的影响

```
// 在Clobber/Modify域标注eax寄存器被修改

1.   int main()
2.   {
3.       int data1 = 10;
4.       int result = 20;
5.
6.       asm ("movl %1, %%eax\n\t"
7.            "addl %%eax, %0"
8.            : "=r"(result)
9.            : "r"(data1), "0"(result)
10.           : "%eax");
11.
12.      printf("The result is %d\n", result); //30
13.      return 0;
14.  }

// 生成的汇编代码

548:    8b 5d f0     mov    -0x10(%ebp),%ebx   //data1
54b:    8b 45 f4     mov    -0xc(%ebp),%eax    //result
54e:    89 c2        mov    %eax,%edx
550:    89 d8        mov    %ebx,%eax
552:    01 c2        add    %eax,%edx
554:    89 55 f4     mov    %edx,-0xc(%ebp)    //result
```

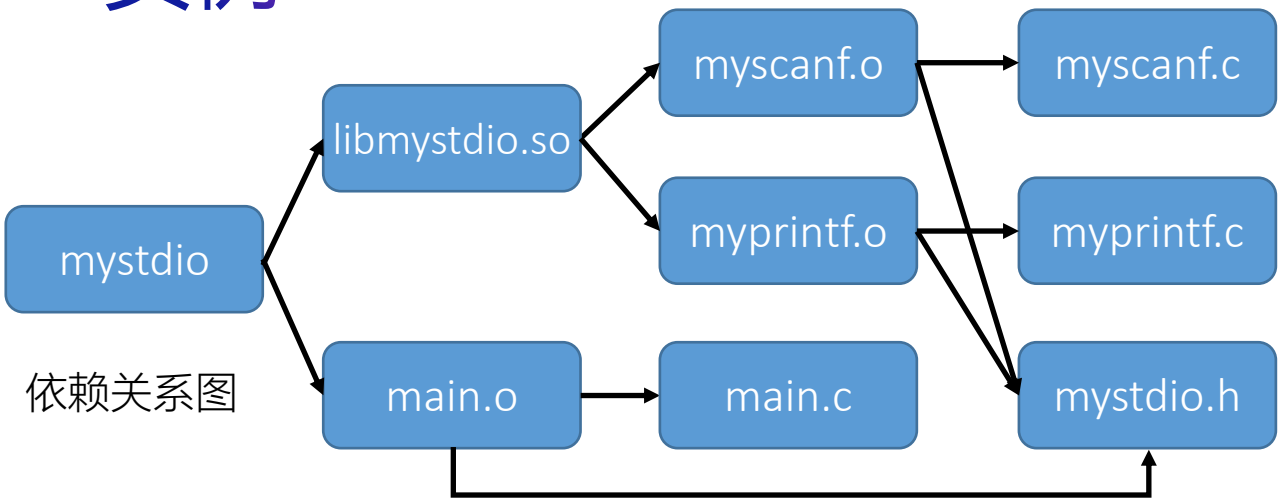
```
// 未在Clobber/Modify域标注eax寄存器被修改

1.   int main()
2.   {
3.       int data1 = 10;
4.       int result = 20;
5.
6.       asm ("movl %1, %%eax\n\t"
7.            "addl %%eax, %0"
8.            : "=r"(result)
9.            : "r"(data1), "0"(result)
10.           );
11.
12.      printf("The result is %d\n", result);  //20
13.      return 0;
14.  }

// 生成的汇编代码

548:    8b 4d f0     mov    -0x10(%ebp),%ecx   //data1
54b:    8b 45 f4     mov    -0xc(%ebp),%eax    //result
54e:    89 c8        mov    %ecx,%eax
550:    01 c0        add    %eax,%eax
552:    89 45 f4     mov    %eax,-0xc(%ebp)    //result
```

2.3 Makefile文件与make程序

- 程序的生成与执行：预处理->编译->汇编->链接->装载->运行
- 大型工程编译：编写编译脚本，指定组件间的依赖关系，编译工具根据编译脚本安排编译流程。
  - 一个大型工程有很多文件组成，每一个文件的改变都会导致工程的重新链接，但不是所有文件都要重新编译
  - 编译脚本记录有文件的依赖信息，在编译时决定需要重新编译哪些文件
- Makefile：是一种纯文本编译脚本，在其中可以指定需要编译哪些文件，哪些先编译，哪些后编译，哪些需要重新编译，最终需要生成怎么样的应用程序
- make：是一个命令，用来解释Makefile脚本，并根据脚本中的指定内容，进行操作

更多内容：https://www.youwei.site/course/kernel/resource/MakefileManual.pdf

实例



依赖关系图

思考：

- main.c发生改变时，哪些文件需要重新编译链接？
- myscanf.c发生改变呢？
- mystdio.h发生改变呢？

```
#编译生成myscanf.o
gcc -fPIC -c myscanf.c

#编译生成myprintf.o
gcc -fPIC -c myprintf.c

#编译生成main.o
gcc -fPIC -c main.c

#生成动态链接库
gcc -shared -o libmystdio.so myscanf.o myprintf.o

#动态链接生成
gcc -o mystdio main.c -L. -lmystdio -Wl,-rpath=.
```

```
1. # Makefile

2. mystdio: main.o libmystdio.so
3.          gcc -o mystdio main.o -L. -lmystdio -Wl,-rpath=.
4. libmystdio.so: myscanf.o myprintf.o
5.          gcc -shared -o libmystdio.so myscanf.o myprintf.o
6. main.o: main.c mystdio.h
7.          gcc -fPIC -c main.c
8. myscanf.o: myscanf.c mystdio.h
9.          gcc -fPIC -c myscanf.c
10.myprintf.o: myprintf.c mystdio.h
11.         gcc -fPIC -c myprintf.c
12.clean:
13.         rm -rf mystdio *.o *.a *.so
```

2.3.1 Makefile的构成

- 主要包含了五种类型的语句
  - 显式规则：显式指定要生成的文件，所依赖的文件，生成的命令
  - 隐式规则：make有自动推导功能，可以让程序员比较简略地书写Makefile
  - 变量定义：变量一般都是字符串，当Makefile被执行时，会进行变量扩展
  - 文件指示：引用另一个Makefile；根据情况指定执行Makefile中的有效部分
  - 注释：注释使用“#”，若Makefile中需要用到“#”，则需要做转义“\#”

```
#sample makefile script
include other.make

CC=gcc
SRCS=fun1.c fun2.c main.c
EXEC=test

all: $(SRCS)
	$(CC) $(SRCS) -o $(EXEC)
clean:
	rm -rf $(EXEC)
```

注释

文件指示，包含其他文件，其他文件中的变量会被包含进来

CC, SRCS, EXEC为变量，都是字符串，使用时会完全被替换

规则部分，变量在被引用时需要加上$()或者${}

2.3.2 Makefile的规则

- 规则解释如何编译文件，make根据依赖关系执行产生或更新目标；

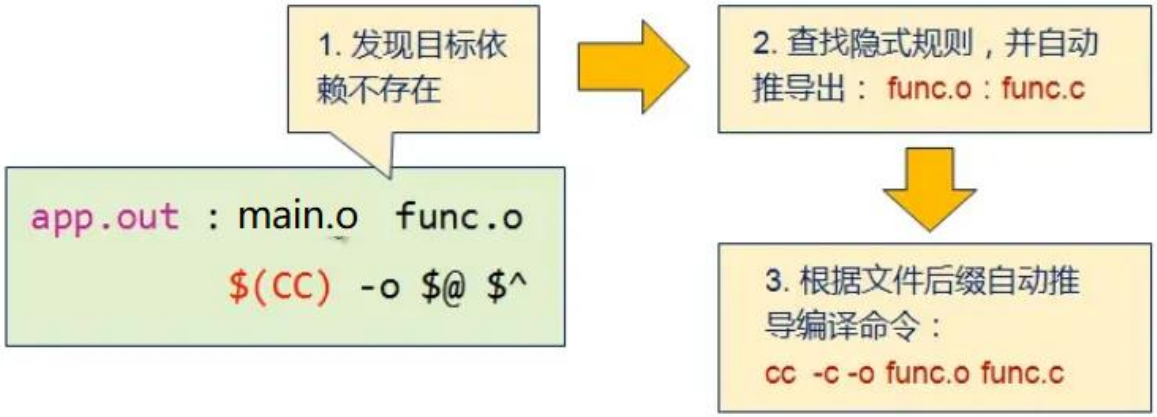
规则也说明如何和何时执行动作。规则的一般模式如下

```
<target>:<depend>
        command1
        command2
        ……
```

- target是一个目标文件，可以是可执行文件或.o文件，也可以是执行动作。
- depend目标的依赖，目标若需要成立，必须有依赖。一个target可以拥有多个depend 。
- command是make执行动作，一个目标依赖关系中可以包含多个命令，但是每个command不能是空格或者其它的字符，只可以一个制表符Tab键。
- 注：若target缺少depend ，那么command会直接被执行。

2.3.2 Makefile的规则

- 隐式规则
  - 当make发现目标的依赖不存在时，将会尝试通过依赖名逐一查找隐式规则，并且通过依赖名推导出可能需要的源文件
  - 隐式规则会使用预定义变量完成编译，改变预定义变量将部分改变隐式规则的行为，当存在自定义规则时，不再使用隐式规则
  - 查看make支持的隐式规则：make -p



2.3.3 Makefile的变量

- 简单赋值（:=），编程语言中常规理解的赋值方式，只对当前语句的变量有效

```
x:=foo
y:=$(x)b
x:=new
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

```
y=>foob
x=>new
```

- 递归赋值（=），赋值语句可能影响多个变量，所有目标变量相关的其他变量都受影响

```
x=foo
y=$(x)b
x=new
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

```
y=>newb
x=>new
```

- 条件赋值（?=），若变量未定义，则使用符号中的值定义变量，否则则该赋值语句无效

```
x:=foo
y:=$(x)b
x?=new
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

```
y=>foob
x=>foo
```

- 追加赋值（+=）原变量用空格隔开的方式追加一个新值

```
x:=foo
y:=$(x)b
x+=$(y)
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

```
y=>foob
x=>foo foob
```

2.3.3 Makefile的变量

- 变量引用：${变量名称}或者$(变量名称)
- 变量替换：foo=$(var:a=b)，将var变量中的a替换成b，并返回给foo

```
SRCS = fun1.c fun2.c main.c
OBJS = $(SRCS: .c=.o)
```

那么变量OBJS值为fun1.o fun2.o main.o

- 预定义变量：Makefile有一些预定义变量， 这些变量有些会在隐式规则中自动带入，有些则是约定俗成的具有特殊意义的变量

```
CC          : 编译器类型
CFLAGS      : 编译选项
LDFLAGS     : 额外链接库
EXEC或APP: 应用程序名
SRCS        : 源代码
OBJS        : 目标文件
```

2.3.4 Makefile的工作原理

```
#sample makefile script
include other.make

CC=gcc
SRCS=fun1.c fun2.c main.c
EXEC=test

all: $(SRCS)
	$(CC) $(SRCS) -o $(EXEC)
clean:
	rm -rf $(EXEC)
```

- 当执行make的时候，make程序从当前目录读入Makefile开始处理第一个非“.”的规则，这称作缺省目标，上述Makefile文件中，缺省目标是all。
- 在执行make的时候，也可指定目标执行，如make clean，那么make会直接读入执行目标clean，跳过all。若执行make all，那么与make功能相同。

2.3.4 Makefile的工作原理

```
#sample makefile script
include other.make

CC=gcc
SRCS=fun1.c fun2.c main.c
EXEC=test

all: $(SRCS)
	$(CC) $(SRCS) -o $(EXEC)
clean:
	rm -rf $(EXEC)
```

- 工程若干个源文件中，某一个文件发生了改变，我们希望只重新编译被修改的那一个文件，其它文件不重新编译。
- make在执行时，会确认所有target是否都是最新的，若target的某个depend的时间比target新，则make会重新根据依赖关系来执行相应命令。

2.3.5 内核模块的Makefile

1、编译进内核的模块

如果需要将一个模块配置进内核，需要makefile中进行配置:

```
obj-y +=foo.o
```

2、编译可加载的模块

所有在配置文件中标记为-m的模块将被编译成可加载模块.ko文件。如果需要将一个模块配置为可加载模块，需要在makefile中进行配置:

```
obj-m +=foo.o
```

3、模块编译以来多个文件

通常的，驱动开发这也会将单独编译在即开发的驱动模块中，当一个驱动模块依赖多个源文件时，需要通过一下方式来指定依赖文件:

```
obi-m +=foo.o
foo-y :=a.o b.o c.o
// foo.o 由a.o,b.o,c.o 生成，然后调用$(LD)-r 将a.o b.o c.o 连接成foo.o文件
```

4、编译选项

在内核态，编译的选项由EXTRA_CFLAGS，EXTRA_AFLAGS和 EXTRA_LDFLAGS修改成了ccflags-y、asflags-y和ldflags-y。这三个变量的值分别对应编译、汇编、链接时的参数。

2.3.5 内核模块的Makefile

```
obj-m += strudent_list.o
CURRENT_PATH := $(shell pwd)
VERSION := v6.6
ARCH := x86_64
LINUX_KERNEL_PATH := /home/user/Kernel/$(VERSION)/$(ARCH)/

all:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean
```

- -C选项表示的是跳转到指定路径下并读取该路径下的Makefile，即转移至(LINUX_KERNEL_PATH)的位置（内核源代码目录），
- M=$(CURRENT_PATH)表示返回到当前路径并继续执行当前的Makefile。
- modules 指明编译的目标是生成内核模块.ko文件。
- 示例：`make VERSION=v5.0 ARCH=i386`