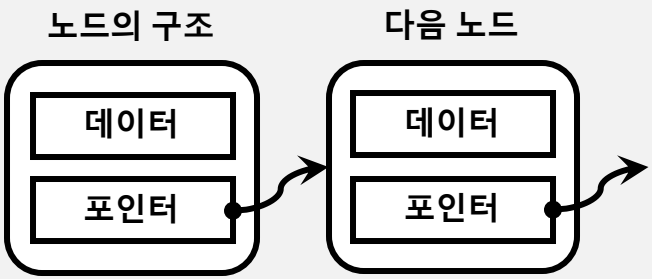
운영체제론 실습

- 생일 목록 불러오는 모듈 프로그래밍

Linked List

- **연결 리스트(Linked List)**는 각 데이터들을 **포인터**로 연결하여 관리하는 구조임
- **노드** : 데이터를 저장하는 **데이터 영역**과 다음 노드를 가리키는 **포인터 영역**으로 구성됨



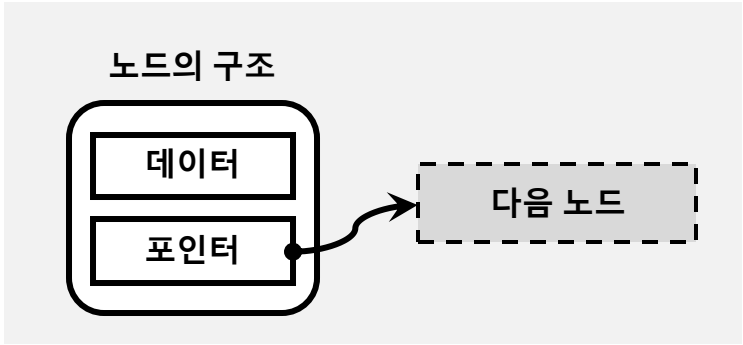
- **Linked List를 사용해서 얻는 이점**
  - ① **동적** 자료구조
  - ② **쉬운 생성과 삭제**
  - ③ 노드의 생성과 삭제가 자유롭기 때문에 **메모리 낭비가 적음**.
  - ④ Linked List를 통해 **다른 자료구조들을 쉽게 구현** 가능.
- **Linked List의 단점**
  - ① 데이터 하나를 표현하기 위해 '**포인터'라는 추가 메모리 사용** (결코 크지 않음)
  - ② 데이터 **탐색하는 시간 복잡도**가 매우 높음 ***O(n)***

Singly Linked List

- 단순 연결 리스트는 다음 노드만을 가리키는 **단방향** 연결 구조임

s_list.c

```
struct Node{
        int data;
        struct Node *next;
};
```



- **함수 예)** 노드의 **생성**

s_list.c

```
node createNode(){
    node new_node;
    new_node = (Node)malloc(sizeof(struct Node));
    new_node ->next = NULL;
    return new_node;
}
```

Doubly Linked List

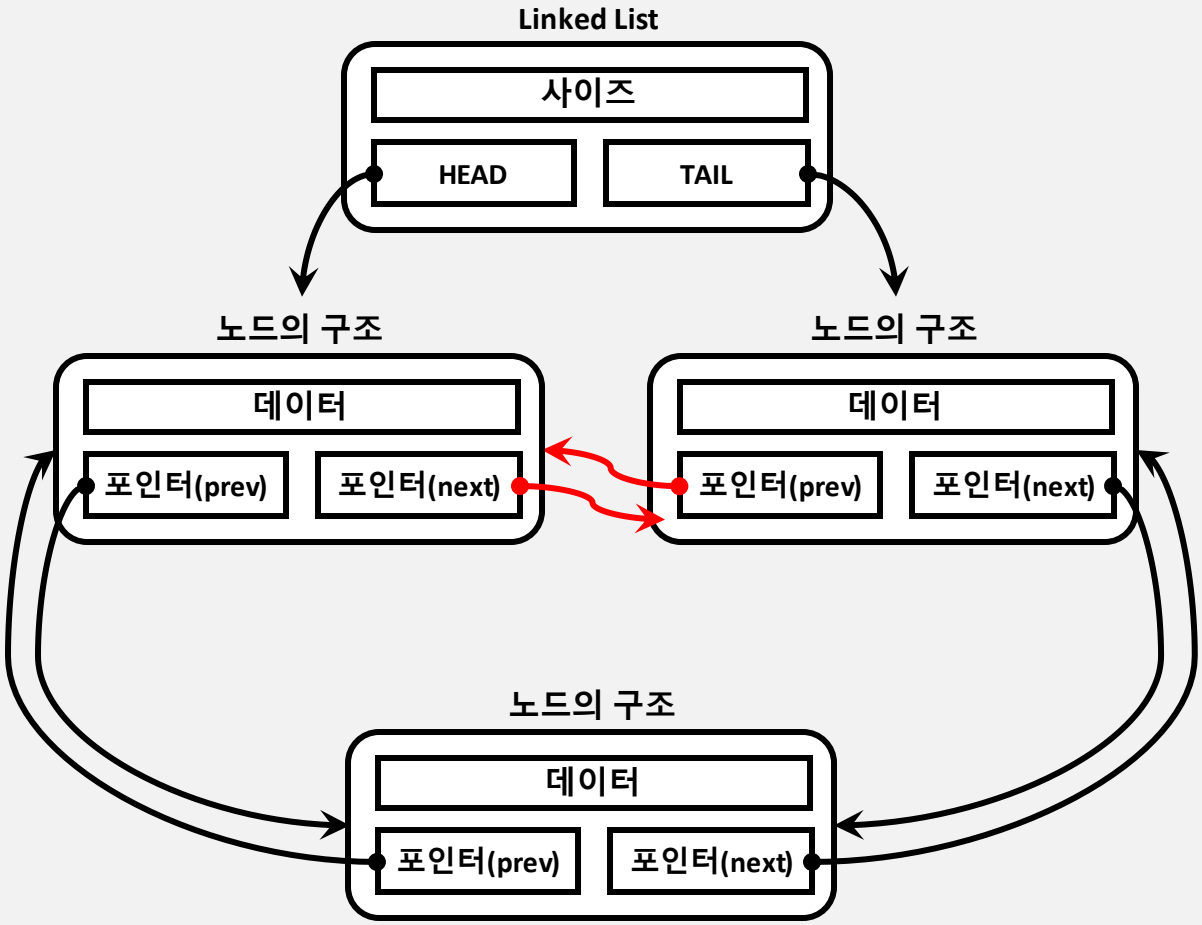
- 이중 연결 리스트는 이전과 다음 노드를 가리키는 **양방향** 연결 구조임

d_list.c

```
struct Node{
        int data;
        struct Node *prev, *next;
};
```

Doubly Circular Linked List

- 이중 원형 연결 리스트는 **처음 노드와 마지막 노드가 상호연결**되어 원형을 이루는 구조임



커널에는 어떻게 구현되어 있는가?

- 커널에는 우리가 알고 있는 Linked List는 어떤 모습을 하고 있을까?

```
$ vi /usr/src/linux-$(uname –r)/include/linux/types.h
```

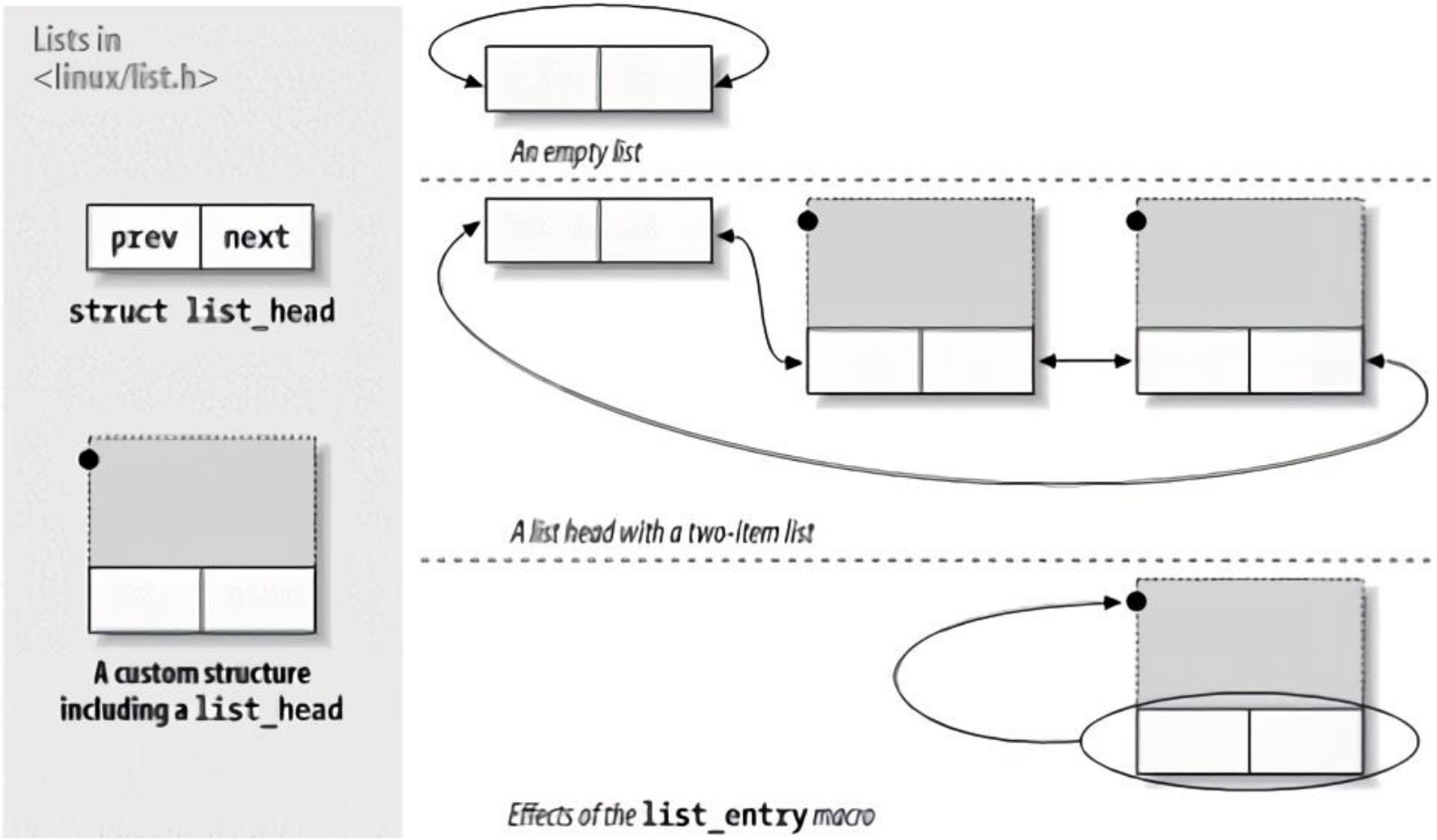
types.h

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

- 이전 노드와 다음 노드를 가리키는 **이중 연결 리스트**임을 알 수 있음
- **이전에 언급해왔던 연결 리스트들과 확연히 다른 점이 보이는가?**

```
struct generic_list{
        void *data;
        struct gerneric_list *prev, *next;
};
```

- **리스트 노드(list_head 구조체)를 사용자가 만든 데이터 안에 넣는 방식.**



생각해보기: 데이터 영역을 어떻게 구현해볼 수 있을까?

- 커널에 구현되어 있는 리스트의 모습은 **Doubly Circular Linked List**임

① 임의의 구조체(struct my_struct) 선언:

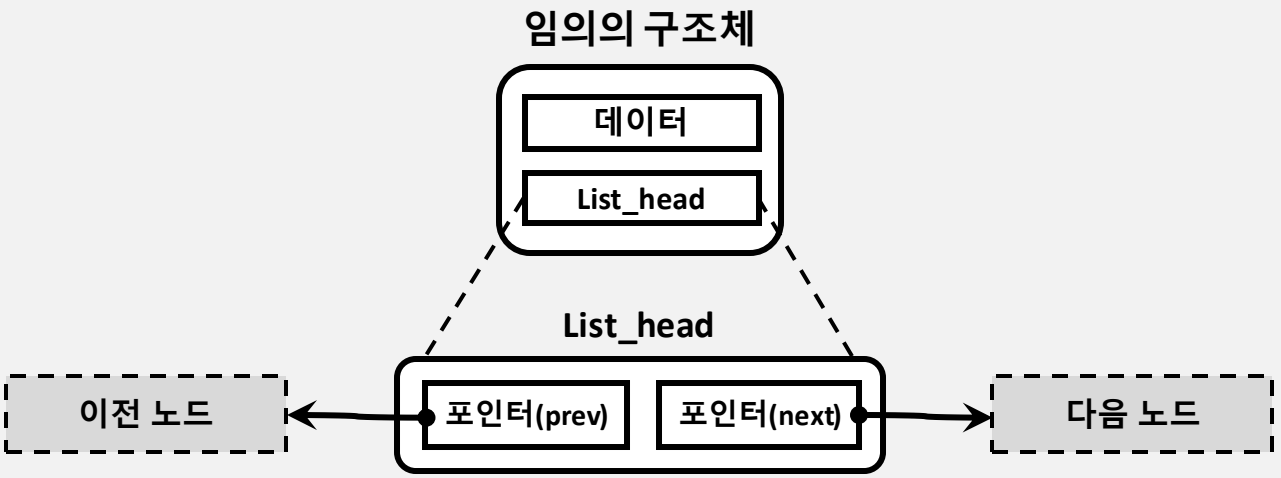
- **struct list_head를 멤버로 넣어줌**

② Head 선언

③ list.h 에서 제공하는 연산을 사용

list_head 인터페이스 사용 방법

- 데이터 영역을 가지는 임의의 구조체를 만들고 list_head를 가리키게 함



임의의 구조체의 예

```
struct my_struct{
        void data; // 저장하고 싶은 데이터
        struct list_head list;
};
```

list.h

- 연결리스트의 구조체, 함수 등이 구현되어 있는 헤더 파일을 살펴보자.

```
$ vi /usr/src/linux-$(uname –r)/include/linux/list.h
```

os@os: /usr/src/linux-5.0.2/include/linux

File Edit View Search Terminal Tabs Help

os@os: ~/os2019/week4/bdlist × | os@os: /usr/src/linux-5.0.2/include/linux ×

```
69 /**
70  * list_add - add a new entry
71  * @new: new entry to be added
72  * @head: list head to add it after
73  *
74  * Insert a new entry after the specified head.
75  * This is good for implementing stacks.
76  */
77 static inline void list_add(struct list_head *new, struct list_head *head)
78 {
79     __list_add(new, head, head->next);
80 }
81
82
83 /**
84  * list_add_tail - add a new entry
85  * @new: new entry to be added
86  * @head: list head to add it before
87  *
88  * Insert a new entry before the specified head.
89  * This is useful for implementing queues.
90  */
@
                                                    69,1          8%
```

list.h

- 기본적인 함수

함수명	목 적
LIST_HEAD(ptr)	리스트 자료구조를 초기화
list_add(new, head)	이전에 만든 리스트에 새로운 entry(list_head *new)를 하나 추가 (맨 앞)
list_add_tail(new, head)	list_add와 동일하나 맨뒤에 추가
list_del(entry)	원하는 entry(list_head)를 삭제
list_empty(head)	비어 있는지 체크 (비면 참)
list_for_each_entry(pos, head, member)	리스트 노드들을 한바퀴 순환하면서, 각 노드들을 참조하는 포인터를 시작주소 지점(entry)으로 옮기는 것
list_for_each_safe(pos, n, head)	entry 의 복사본을 사용함으로써 수행 시 해당 자료가 삭제되더라도 오류가 나지 않게 하는 것

데이터 생성

- Kmalloc을 통해 struct에 메모리공간 할당한다.
  - kmalloc 는 커널 내부에 페이지 크기보다 작은 크기의 메모리 공간을 할당할 때 사용한다.
  - GFP_KERNEL : 보통 커널 RAM 메모리를 할당한다.

- 사용방법

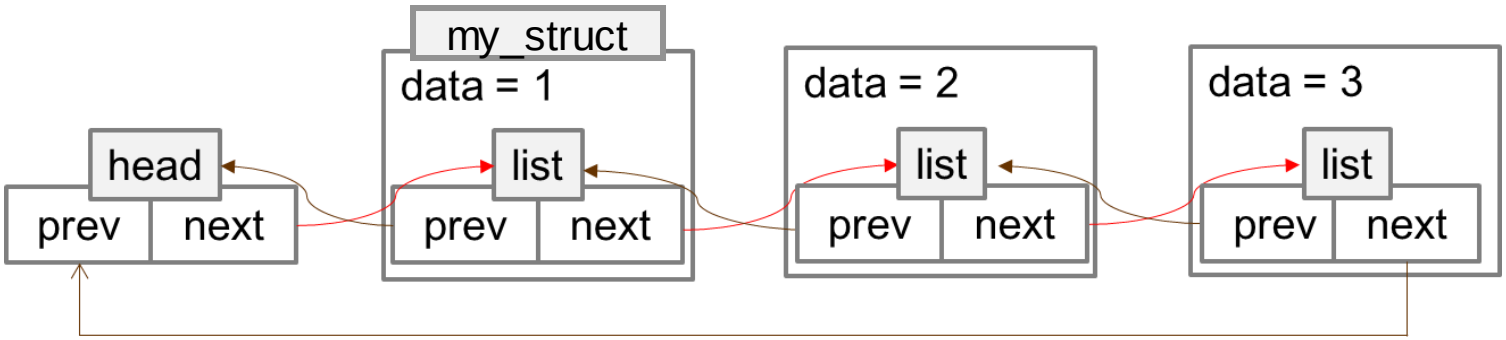
```
struct my_struct{ … }
new_mem_space = kmalloc(sizeof(*my_struct), GFP_KERNEL);
```

데이터 삽입

- **list_add_tail** 함수 사용

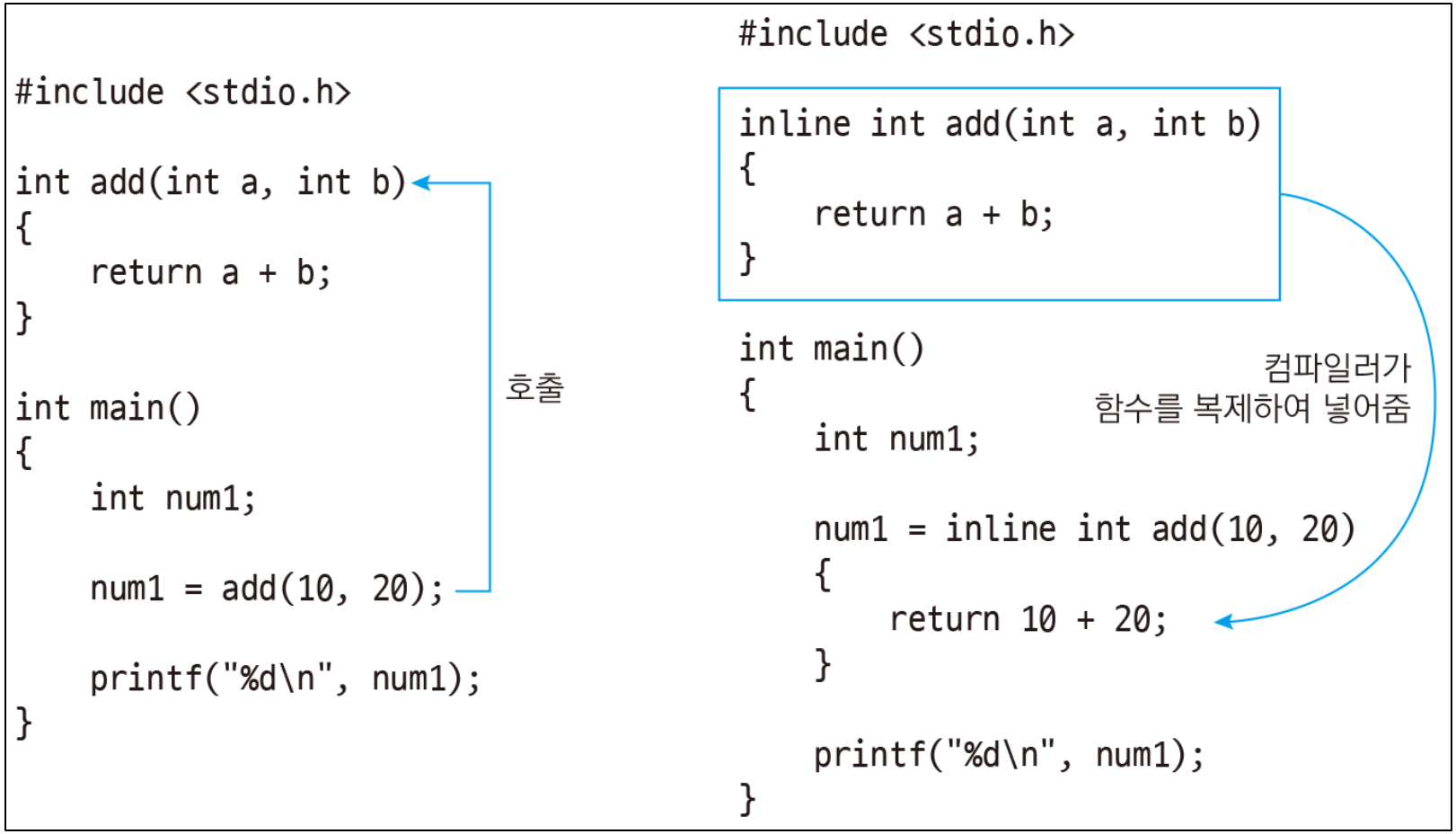
/ include / linux / list.h

```
83  /**
84   * list_add_tail - add a new entry
85   * @new: new entry to be added
86   * @head: list head to add it before
87   *
88   * Insert a new entry before the specified head.
89   * This is useful for implementing queues.
90   */
91  static inline void list_add_tail(struct list_head *new, struct list_head *head)
92  {
93          __list_add(new, head->prev, head);
94  }
```



inline 함수

- 실행 과정이 일반 함수와 크게 다르지 않다.
- 컴파일러는 함수를 사용하는 부분에 함수의 코드를 복제해서 넣어준다.



데이터 출력

- list_for_each_entry라는 매크로 함수를 사용(반복적으로 탐색하며 주어진 타입을 확인)

/ include / linux / list.h

```
510  /**
511   * list_for_each_entry  -       iterate over list of given type
512   * @pos:        the type * to use as a loop cursor.
513   * @head:       the head for your list.
514   * @member:     the name of the list_head within the struct.
515   */
516  #define list_for_each_entry(pos, head, member)                          \
517          for (pos = list_first_entry(head, typeof(*pos), member);        \
518               &pos->member != (head);                                    \
519               pos = list_next_entry(pos, member))
520
```

- 본 함수를 모듈 생성 시 구현하고, 추가할 내용 : **printk**(출력할 구조체의 내용)

데이터 삭제(1)

- list_for_each_safe 매크로 함수 사용 (반복적으로 탐색하며 노드마다 함수 수행)

/ include / linux / list.h

```
489  /**
490   * list_for_each_safe - iterate over a list safe against removal of list entry
491   * @pos:        the &struct list_head to use as a loop cursor.
492   * @n:          another &struct list_head to use as temporary storage
493   * @head:       the head for your list.
494   */
495  #define list_for_each_safe(pos, n, head) \
496          for (pos = (head)->next, n = pos->next; pos != (head); \
497                  pos = n, n = pos->next)
498
```

- 본 매크로 함수에 추가할 내용
  - printk(출력할 구조체의 내용)
  - list_del(삭제할 구조체의 list_head의 주소값)
  - kfree(삭제할 구조체 메모리의 포인터)

매크로 함수

- 매크로 함수 예제

```
#define ADD(a , b) a + b
```

- 코드 내부에 다음과 같이 매크로 함수를 사용했을 경우

```
...
int result = ADD(2,3);
...
```

- 연산을 수행하기 이전에 **전처리기**에 의해 코드가 그대로 치환됨

```
...
int result = 2 + 3;
...
```

실습: 생일 목록을 불러오는 모듈 프로그래밍

- **TODO:**
  - 생일 데이터를 가지는 구조체를 만든다.
  - 생일 데이터들끼리 커널의 연결리스트를 통해 연결한다.
  - 이 데이터들을 전부 출력한다.

실습: 스켈레톤 코드 (1) (다운로드: 클릭)

os@os: ~/os2019/week4/bdlist

File Edit View Search Terminal Help

```c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/list.h>
#include <linux/module.h>
#include <linux/slab.h>

struct birthday {
  int day;
  int month;
  int year;
  struct list_head list;
};

LIST_HEAD(birthday_list);

// kernel memory allocation & fill the data at struct birthday
struct birthday *createBirthday(int day, int month, int year) {
    /* Write your code */
}

// insert each list_head to birthday_list
int simple_init(void) {
    printk("Loading Module: BDLIST.....");

    /* Write your code */
}
```

실습: 스켈레톤 코드 (2)

```
27
28 void simple_exit(void) {
29     
30     /* Write your code */
31
32     printk("Removing Module: BDLIST....");
33 }
34
35 module_init(simple_init);
36 module_exit(simple_exit);
37
38 MODULE_LICENSE("GPL");
39 MODULE_DESCRIPTION("Simple Module");
40 MODULE_AUTHOR("My Name");
```

실습: 결과화면

```
[ 2558.978567] Loading Module: BDLIST......
[ 2558.978573] OS Module: Day 13.4.1987
[ 2558.978577] OS Module: Day 14.1.1964
[ 2558.978581] OS Module: Day 2.6.1964
[ 2558.978583] OS Module: Day 13.8.1986
[ 2558.978586] OS Module: Day 10.6.1990
[ 2585.156063] OS Module: Removing 13.4.1987
[ 2585.156068] OS Module: Removing 14.1.1964
[ 2585.156071] OS Module: Removing 2.6.1964
[ 2585.156074] OS Module: Removing 13.8.1986
[ 2585.156077] OS Module: Removing 10.6.1990
[ 2585.156079] Removing Module: BDLIST....
os@os:~/os2019/week4/bdlist$
```

*생일데이터는 임의로 작성해도 무관

수고하셨습니다.

- 다음 시간: **UNIX 쉘과 History 기능 구현**