

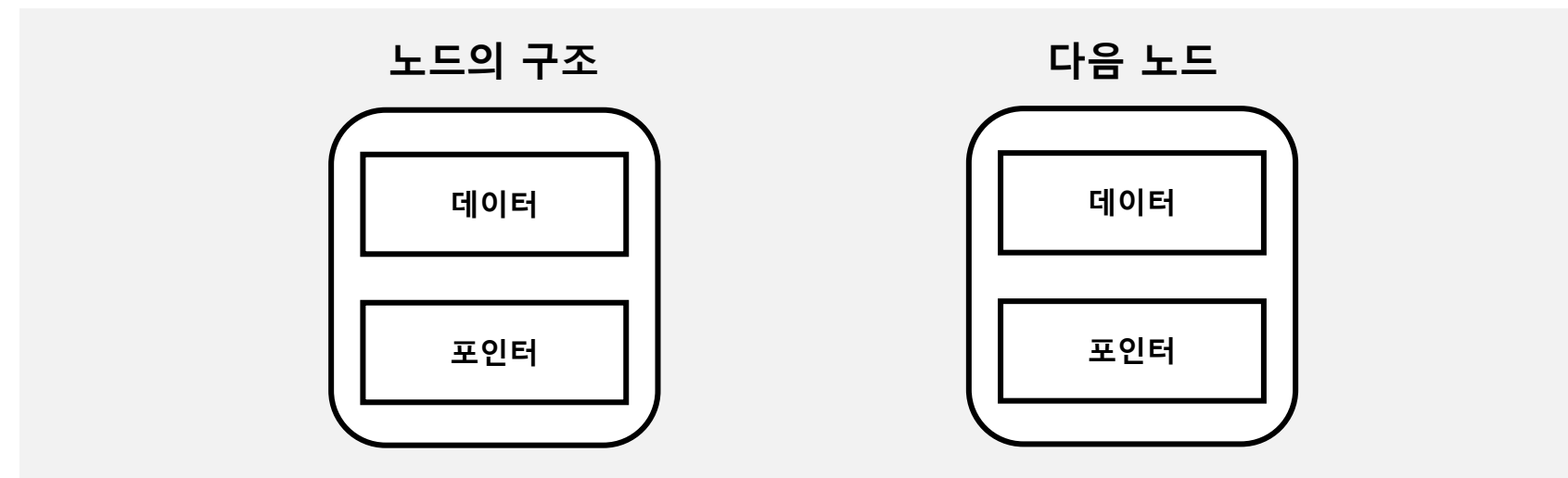
운영체제론 실습 3주차

CPS LAB

생일 목록 불러오는 모듈 프로그래밍

Linked List

- 연결 리스트(Linked List)는 각 데이터들을 포인터로 연결하여 관리하는 구조이다.
- 노드 : 데이터를 저장하는 데이터 영역과 다음 노드를 가리키는 포인터 영역으로 구성된다.



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

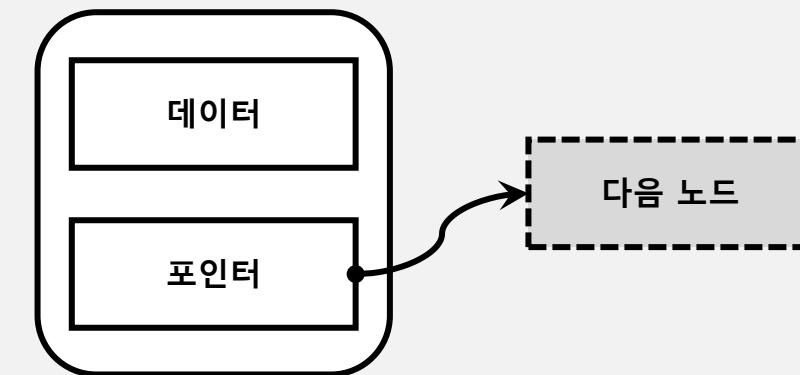
Singly Linked List

- 단일 연결 리스트는 다음 노드만을 가리키는 단방향 연결 구조임

s_list.c

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

노드의 구조



- 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

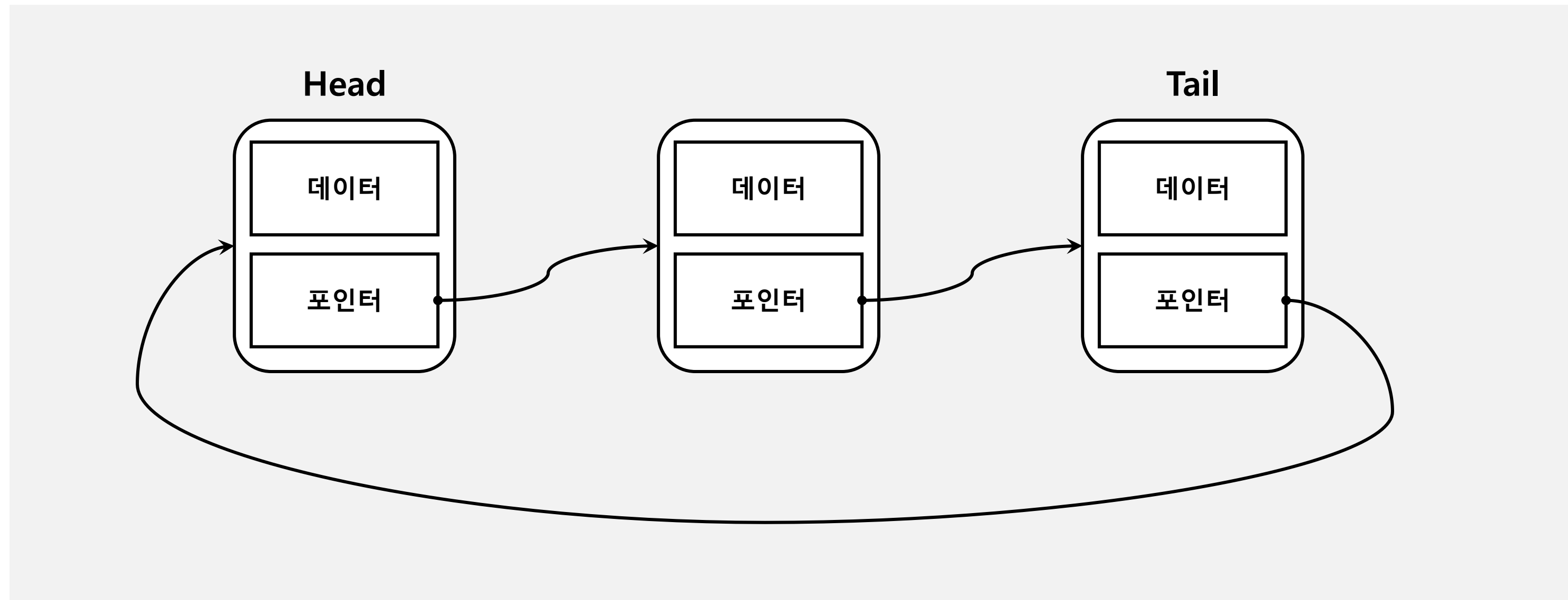


d_list.c

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

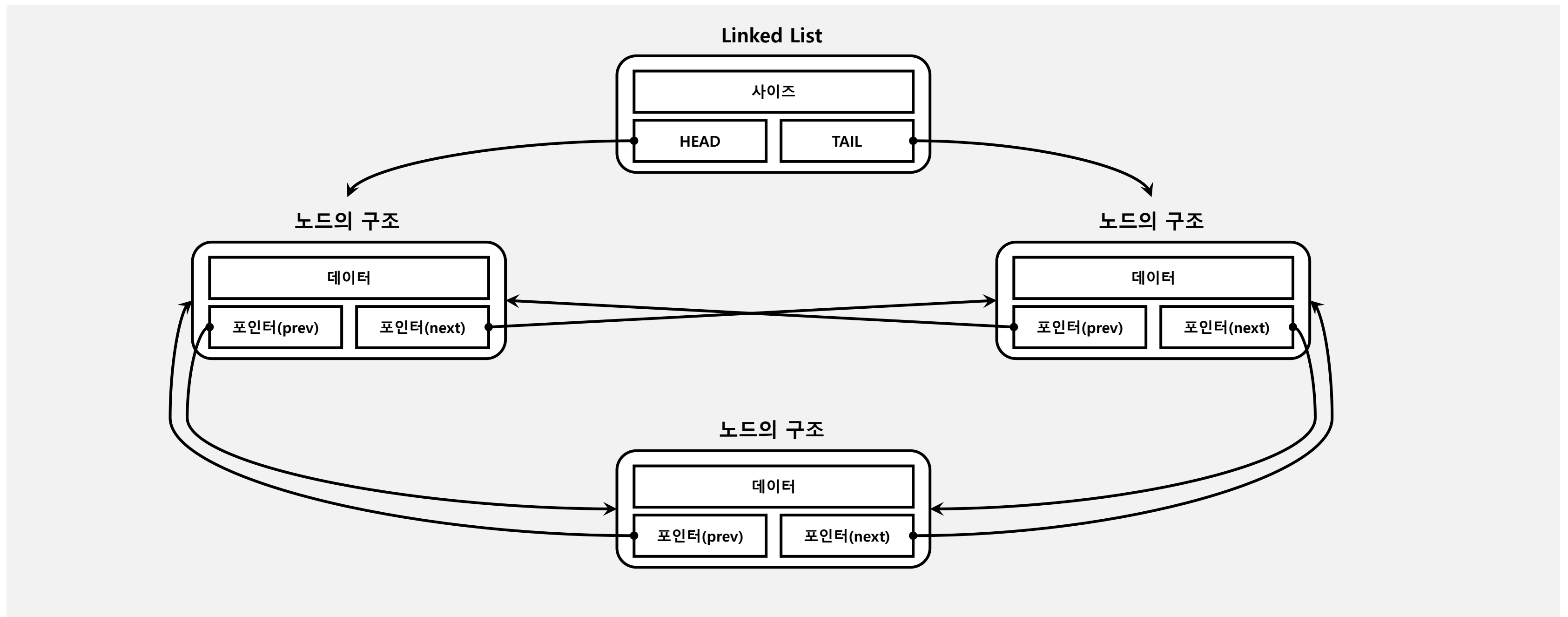
Circular Linked List

- 원형 연결 리스트는 마지막 노드가 첫 번째 노드를 가리켜서, 원형을 이루는 구조



Doubly Circular Linked List

- 이중 원형 연결 리스트는 처음 노드와 마지막 노드가 상호 연결되어 원형을 이루는 구조임



커널에는 어떻게 구현되어 있는가?

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

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

types.h

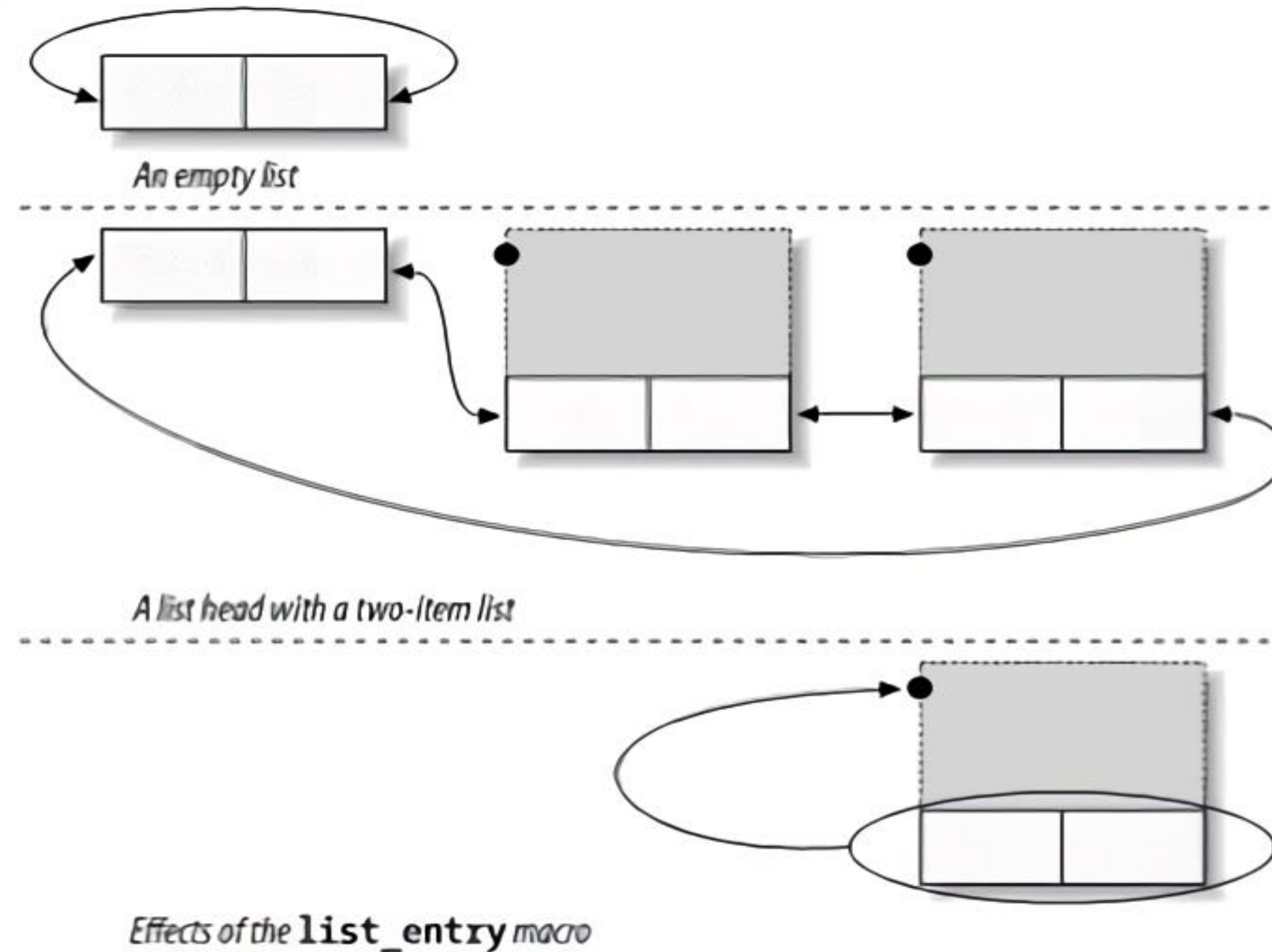
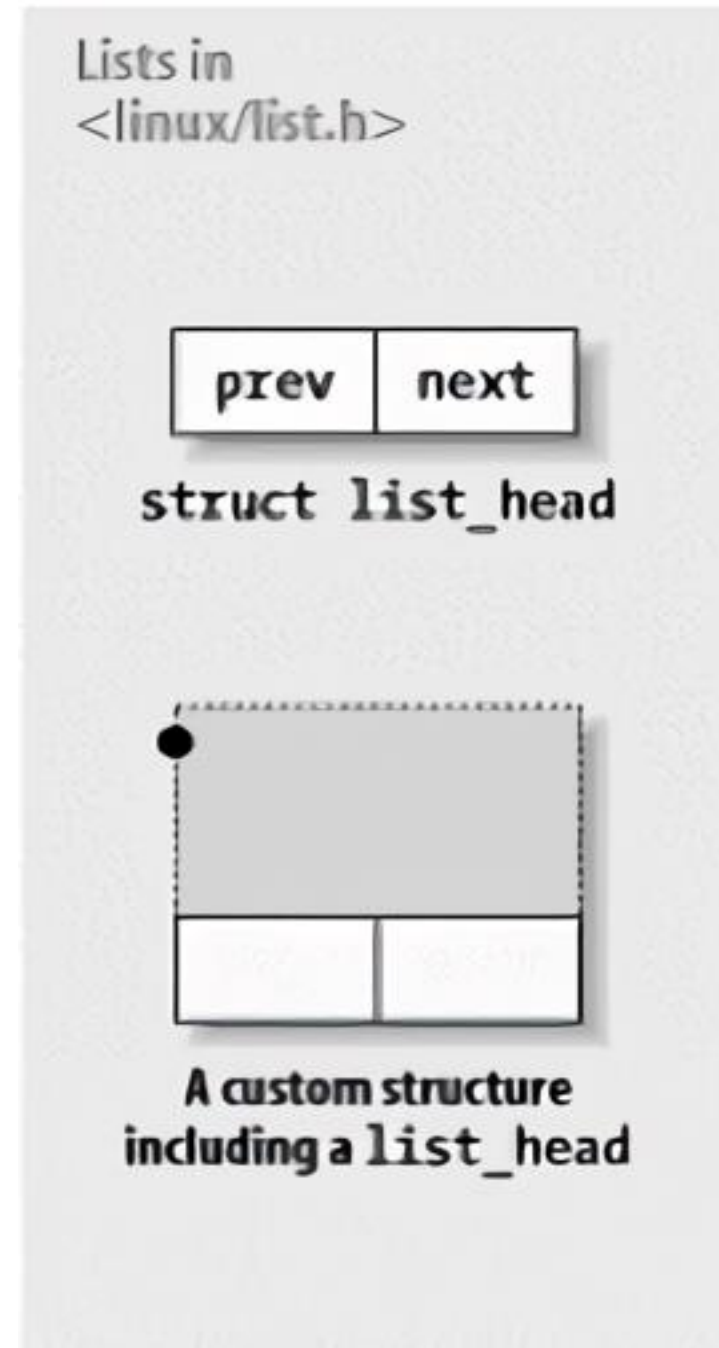
```
struct list_head{  
    struct list_head *prev, *next;  
};
```

- 이전 노드와 다음 노드를 가리키는 이중 연결 리스트임을 알 수 있음
- 이전에 언급해왔던 연결 리스트들과 확연히 다른 점이 보이나요?

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

커널에는 어떻게 구현되어 있는가?

- 리스트 노드(list head 구조체)를 사용자가 만든 데이터 안에 넣는 방식.

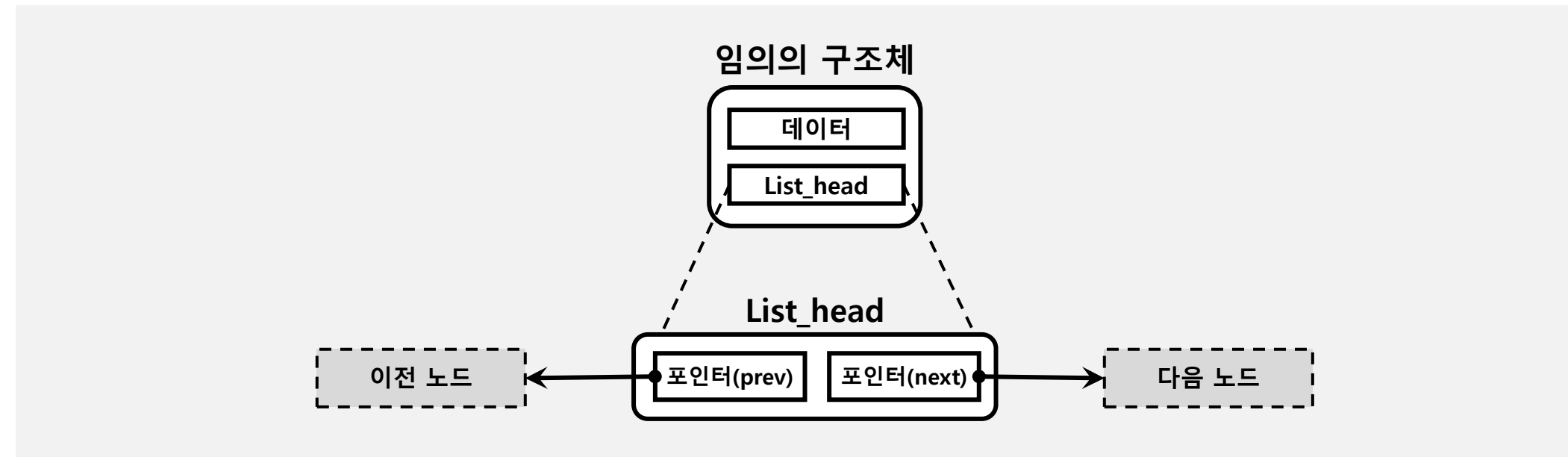


데이터 영역을 어떻게 구현해볼 수 있을까?

- 커널에 구현되어 있는 리스트의 모습은 Doubly Circular Linked List임
-
- ① 임의의 구조체(struct my_struct) 선언:
 - **struct list_head를 멤버로 넣어 줌**
 - 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

- 연결 리스트의 구조체, 함수 등이 구현되어 있는 header file

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

```
os@os-virtual-machine: ~
File Edit View Search Terminal Help
75
76 /**
77  * list_add - add a new entry
78  * @new: new entry to be added
79  * @head: list head to add it after
80  *
81  * Insert a new entry after the specified head.
82  * This is good for implementing stacks.
83  */
84 static inline void list_add(struct list_head *new, struct list_head *head)
85 {
86     __list_add(new, head, head->next);
87 }
88
89
90 /**
91  * list_add_tail - add a new entry
92  * @new: new entry to be added
93  * @head: list head to add it before
94  *
95  * Insert a new entry before the specified head.
96  * This is useful for implementing queues.
97  */
```

list.h

- 기본적인 함수

함수명	목 적
LIST_HEAD(ptr)	ptr이란 이름의 list_head를 정의 후 리스트 자료구조를 초기화
list_add(struct list_head *new, struct list_head *head);	이전에 만든 리스트에 새로운 entry(list_head)를 맨 앞에 추가
list_add_tail(struct list_head *new, struct list_head *head);	list_add와 동일하나 맨 뒤에 추가
list_del(struct list_head *entry);	원하는 entry(list_head)를 삭제
list_empty(struct list_head *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

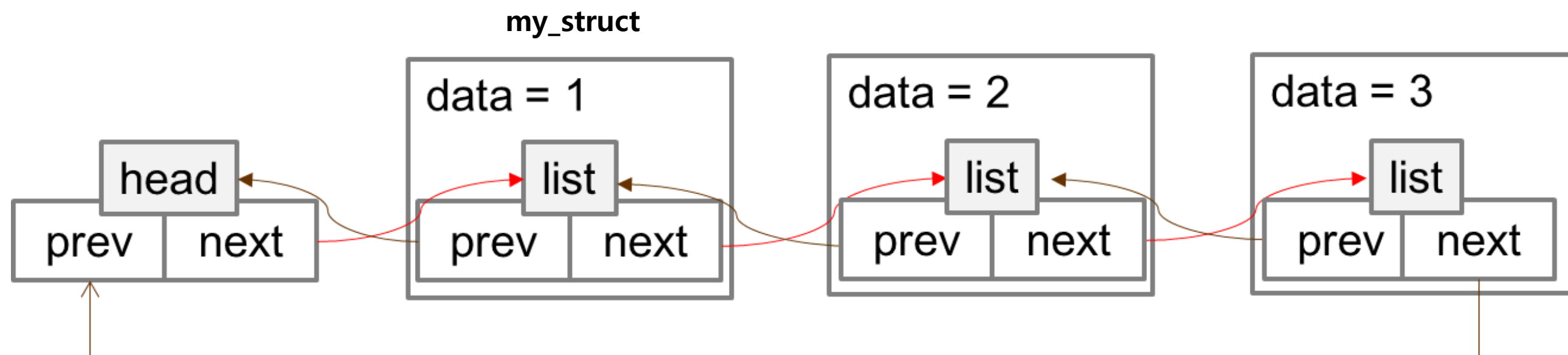
- kmalloc을 통해 물리 메모리 공간 할당
 - kmalloc은 커널 내부에 페이지 크기보다 작은 크기의 메모리 공간을 할당할 때 사용함
 - GFP_KERNEL : 보통 커널 RAM 메모리를 할당함
- 사용 방법

```
struct my_struct *struct1;  
struct1 = kmalloc(sizeof(struct my_struct), GFP_KERNEL);
```

데이터 삽입

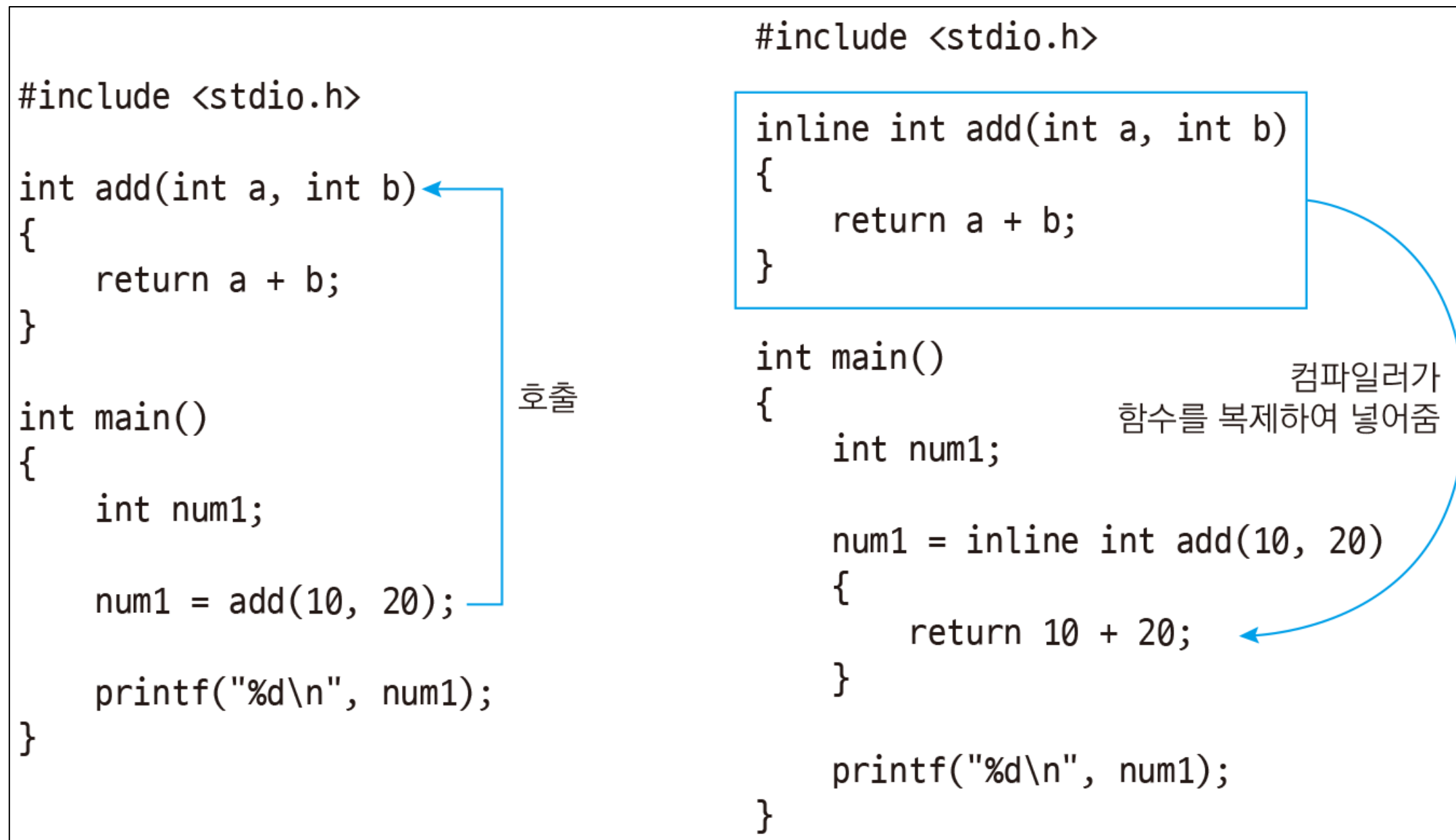
- `list_add_tail()` 함수 사용

```
os@os-virtual-machine: ~  
File Edit View Search Terminal Help  
87 }  
88  
89  
90 /**  
91  * list_add_tail - add a new entry  
92  * @new: new entry to be added  
93  * @head: list head to add it before  
94  *  
95  * Insert a new entry before the specified head.  
96  * This is useful for implementing queues.  
97  */  
98 static inline void list_add_tail(struct list_head *new, struct list_head *head)  
99 {  
100     __list_add(new, head->prev, head);  
101 }  
102
```



inline 함수

- 실행 과정이 일반 함수와 크게 다르지 않음
- 컴파일러는 함수를 사용하는 부분에 함수의 코드를 복제해서 넣어 줌



데이터 출력

- `list_for_each_entry()` 매크로 함수 사용 (반복적으로 탐색하며 주어진 타입을 확인)

```
os@os-virtual-machine: ~
File Edit View Search Terminal Help
620
621 /**
622  * list_for_each_entry - iterate over list of given type
623  * @pos: the type * to use as a loop cursor.
624  * @head: the head for your list.
625  * @member: the name of the list_head within the struct.
626  */
627 #define list_for_each_entry(pos, head, member) \
628     for (pos = list_first_entry(head, typeof(*pos), member); \
629         !list_entry_is_head(pos, head, member); \
630         pos = list_next_entry(pos, member))
631
```

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

데이터 삭제

- **list_for_each_safe()** 매크로 함수 사용 (반복적으로 탐색하며 node마다 함수 수행)

```
os@os-virtual-machine: ~
File Edit View Search Terminal Help
590
591 /**
592  * list_for_each_safe - iterate over a list safe against removal of list entry
593  * @pos:          the &struct list_head to use as a loop cursor.
594  * @n:           another &struct list_head to use as temporary storage
595  * @head:        the head for your list.
596  */
597 #define list_for_each_safe(pos, n, head) \
598     for (pos = (head)->next, n = pos->next; pos != (head); \
599         pos = n, n = pos->next)
600
```

- 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 전처리기에 의해 코드가 그대로 치환됨

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

※ inline 함수와 매크로 함수의 차이점

<https://ko.gadget-info.com/difference-between-inline>

실습 : Skeleton code (1)

```
os@os-virtual-machine: ~/week3/bdlist
File Edit View Search Terminal Help
#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;
};

static LIST_HEAD(birthday_list);

struct birthday *createBirthday(int day, int month, int year) {
    /* 1. TODO: 생일을 위한 메모리를 할당하고, 인자들을 채워 생일을 완성하세요. */
}

int simple_init(void) {
    printk("INSTALL MODULE: bdlist\n");

    /* 2. TODO: 생일 목록을 하나씩 생성하는대로 연결리스트에 연결시키세요(노드 삽입). */

    /* 3. TODO: 완성된 연결리스트를 탐색하는 커널 함수를 사용하여 출력하세요. */

    return 0;
}
```

실습 : Skeleton code (2)

```
/* 모듈을 제거할 때는 생성한 연결 리스트도 하나씩 제거하며 끝내도록 하세요. */
void simple_exit(void) {
    /* 제거를 하기 전에 리스트가 "비어있을 경우"에 대한 예외처리 */
    if(list_empty(&birthday_list)) {
        printk("List is Empty\n");
        return;
    }

    /* 4. TODO: 이제 본격적으로 연결리스트를 탐색하면서 하나씩 제거하도록 하시면 됩니다. */

    /* 다만, 제거를 하면서 연결리스트를 탐색하면 문제가 생길 것 같은데 어떤 방법으로 해결 가능한지
    생각해 보세요. */

    printk("REMOVE MODULE: bdlist\n");
}

module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("make a list of birthdays and print");
MODULE_AUTHOR("이름_학번"); // 이름_학번 형식으로 적어서 제출하세요.
```

※ LMS 강의자료실에서 Skeleton Code와 list.h 함수 예제 코드 다운로드 가능

실습 : 결과 화면

```
[ 190.823673] INSTALL MODULE: bdlist
[ 190.823674] OS Module: Day 13.4.1987
[ 190.823675] OS Module: Day 14.1.1964
[ 190.823676] OS Module: Day 2.6.1964
[ 190.823676] OS Module: Day 13.8.1986
[ 190.823677] OS Module: Day 10.8.1996
[ 237.542593] OS Module: Removing 13.4.1987
[ 237.542595] OS Module: Removing 14.1.1964
[ 237.542596] OS Module: Removing 2.6.1964
[ 237.542597] OS Module: Removing 13.8.1986
[ 237.542598] OS Module: Removing 10.8.1996
[ 237.542598] REMOVE MODULE: bdlist
```

※ 생일 데이터는 임의로 작성해도 무관

감사합니다.

CPS LAB

Lim Jiseoup

jseoup@hanyang.ac.kr