KWANGWOON UNIVERSITY

2024년 2학기 **운영체제실습** 6주차

Task Management

System Software Laboratory
School of Computer and Information Engineering
Kwangwoon Univ.

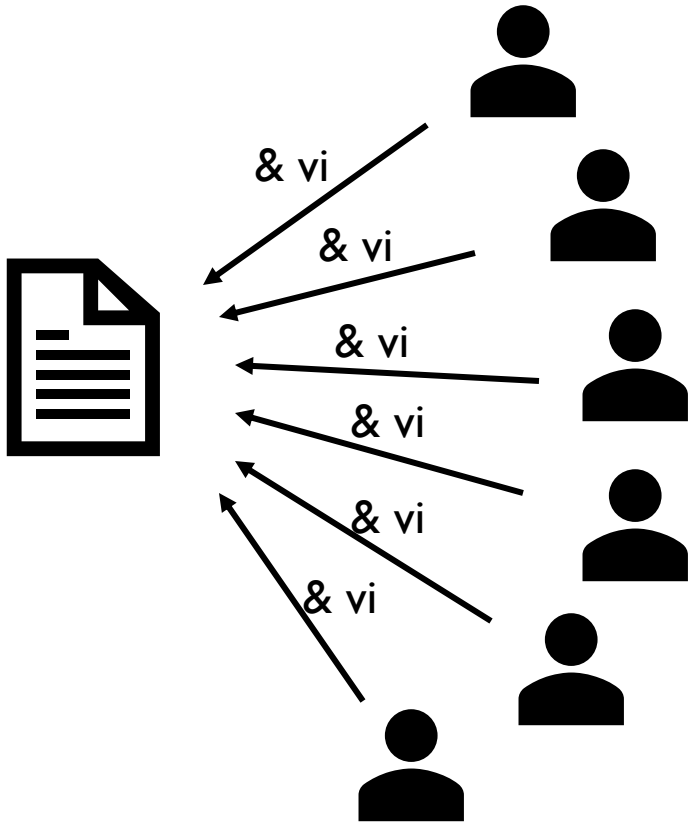
Contents

- Process와 Process Descriptor의 이해
- task_struct 구조체
- Lab. 1
- 프로세스 계보 (family)

프로세스

- **6명의 사용자가 vi 프로그램을 동시에 사용하는 경우**
  - 각각 다른 프로세스가 6개 존재
  - 각각 다른 task_struct 구조체가 6개 존재

- **리눅스 코드에서 프로세스는 Task 또는 Thread라 부르기도 한다.**



Process와 Process Descriptor의 이해

- **Process Descriptor**
  - 변화하는 process의 모든 정보를 담고 있는 자료구조
    - General Term : **PCB** (Process Control Block)
    - Linux Specific : **task_struct** 라는 자료 구조를 사용
  - 즉, 리눅스에서 Process descriptor(프로세스 기술자)는 task_struct 자료 구조
  - 커널은 각 프로세스가 무엇을 하고 있는지 명확히 알아야 한다.
    - E.g.
      - 프로세스의 ID 및 우선 순위
      - 프로세스가 실행 상태
      - 프로세스가 할당 되어있는 주소 공간
      - 다룰 수 있는 파일
      - ……

task_struct 구조체 (1/11)

- **구조체를 통한 정보 관리**
  - 프로세스가 생성되면 task_struct 구조체를 통해 프로세스의 모든 정보를 저장하고 관리
    - 모든 태스크들에게 하나씩 할당
    - include/linux/sched.h
    - 태스크 ID, 상태 정보, 가족 관계, 명령, 데이터, 시그널, 우선순위, CPU 사용량 및 파일 디스크립터 등 생성된 태스크의 모든 정보를 가짐
    - task_struct ***current**
      - 현재 실행되고 있는 태스크를 가리키는 매크로
      - in <include/asm-generic/current.h>

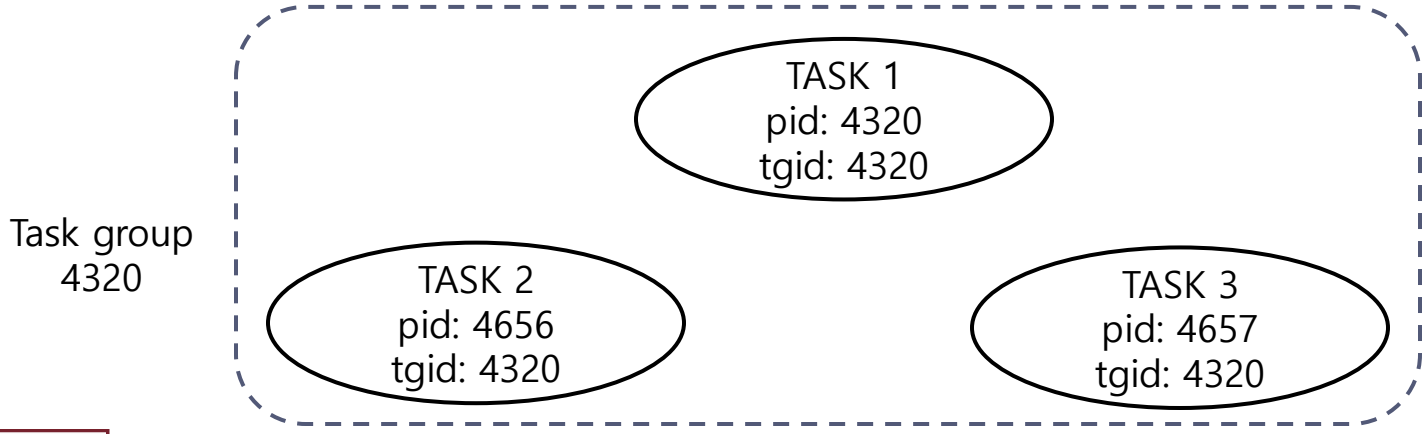
 # task_struct 구조체

task_struct 구조체 (3/11)

- **태스크 식별 정보에 관련된 변수, 함수들**

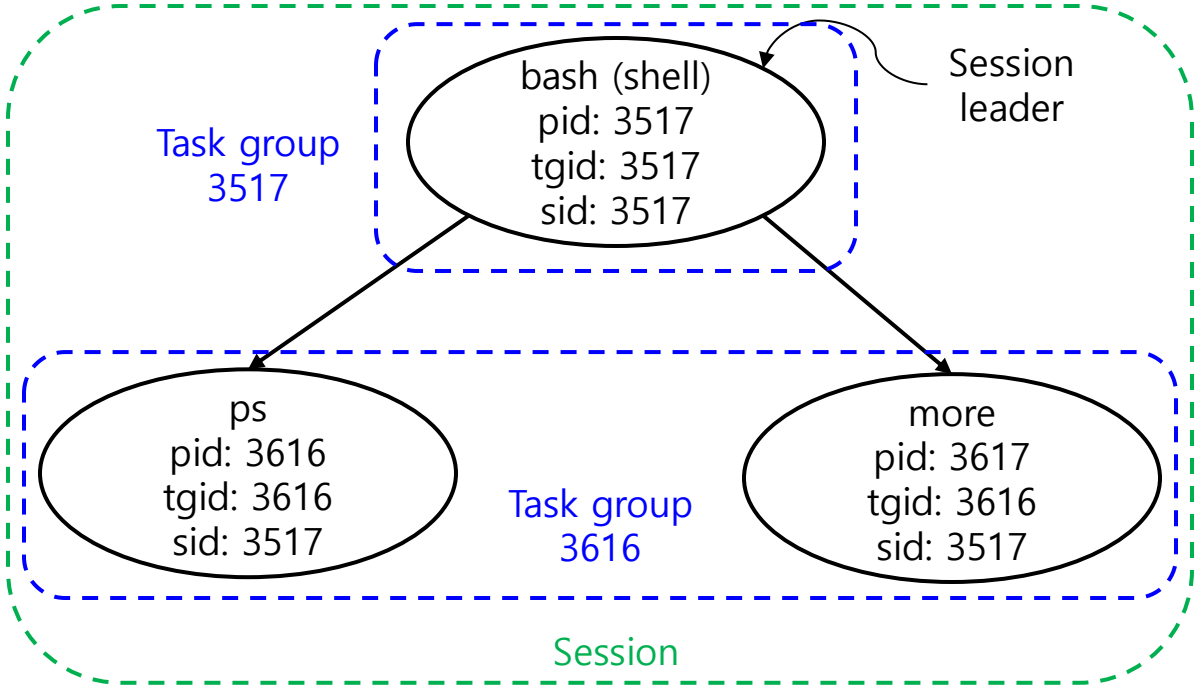
```
pid_t pid;                                  // Thread(Lightweight Process) ID
pid_t tgid;                                 // Thread Group(Process) ID
struct hlist_node pid_links[PIDTYPE_MAX];
```

  - 리눅스 커널에서의 태스크 식별
    - 프로세스와 스레드 모두 task_struct로 관리 (공유, 접근제어의 차이)
    - 시스템에 존재하는 태스크를 구분하기 위해, 각 태스크에 pid를 할당
    - 한 프로세스 내에서 생성된 스레드들은 동일한 tgid를 가지는 그룹을 형성하고, 각각의 스레드들은 유일한 pid를 가짐



task_struct 구조체 (4/11)

- **태스크 식별 정보에 관련된 변수, 함수들**
  - 태스크 그룹과 세션 리더의 예
    - 사용자가 콘솔로 로그인하여 shell을 띄웠다고 가정
      - shell도 하나의 태스크이므로 자신의 PID를 가지며, 하나의 태스크 그룹을 형성
      - #ps | more 명령을 실행하면 각 명령에 대해 각각 PID 값을 부여 받고, 이 두 명령은 하나의 태스크 그룹을 형성
      - 두 개의 태스크 그룹은 하나의 세션을 이루며 이때 shell은 세션 리더가 됨



task_struct 구조체 (6/11)

- **태스크들에 대한 사용자의 접근 제어에 관련된 변수, 함수, 매크로**
  - 태스크가 생성되면 사용자의 ID와 사용자가 속한 그룹의 ID가 등록됨

```
struct cred *cred;
gid_t GROUP_AT(struct group_info *gi, int i);
```

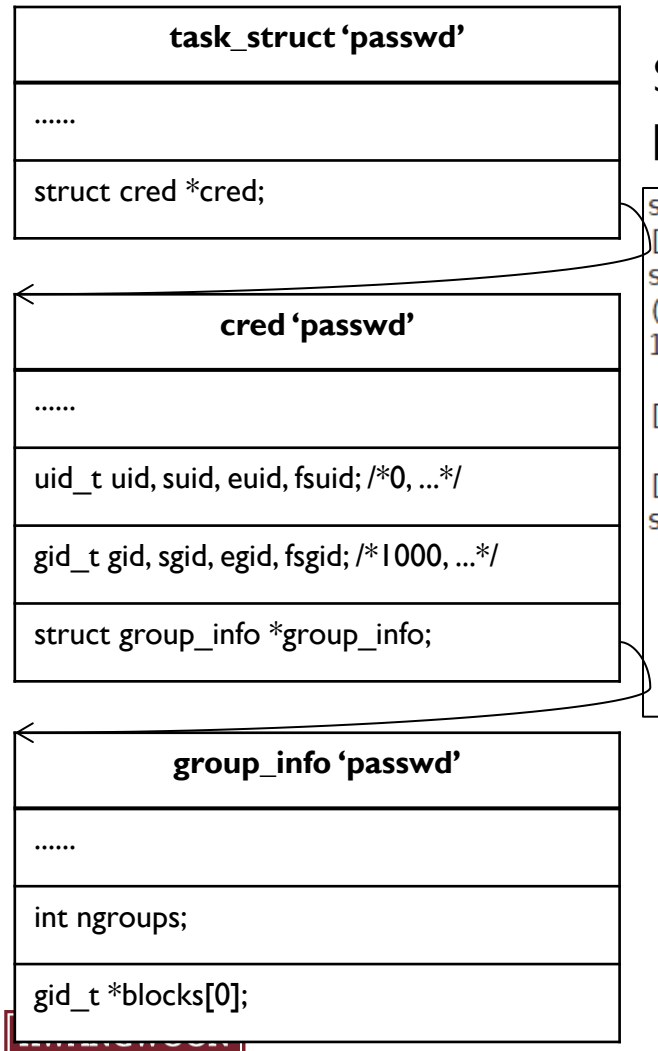
  - struct cred
    - 권한 관련 변수들이 저장된 구조체.
    - (uid, suid, euid, fsuid, gid sgid, egid, fguid)

  - struct group_info
    - 그룹들의 정보를 가진 구조체.

```
for(i=0;  i < current->cred->group_info->ngroup; ++i )
{
        gid = GROUP_AT(current->cred->group_info, i);
}
```

task_struct 구조체 (7/11)

- **task_struct, cred, group_info의 관계**



$ sudo ps –o pid,ppid,uid,suid,euid,fsuid,gid,sgid,egid,fsgid,cmd

```
sslab@sslab:~$ passwd &
[2] 5306
sslab@sslab:~$ Changing password for sslab.
(current) UNIX password: 123
123: command not found

[2]+  Stopped                 passwd

[2]+  Stopped                 passwd
sslab@sslab:~$ sudo ps -o pid,ppid,uid,suid,euid,fsuid,gid,sgid,egid,fs
  PID  PPID   UID  SUID  EUID FSUID   GID  SGID  EGID FSGID CMD
 5294  5196     0     0     0     0  1000  1000  1000  1000 passwd
 5306  5196     0     0     0     0  1000  1000  1000  1000 passwd
 5309  5196     0     0     0     0  1000  1000  1000  1000 sudo ps -o
 5310  5309     0     0     0     0     0     0     0     0 ps -o pid,p
```

suid (saved uid) : 권환 전환을 지원하는데 사용, uid 값을 저장하고 있다

euid (effective uid) : 프로세스가 파일에 대해 가지는 권한
파일에 접근 시 euidd를 통해 파일 접근 허용 여부 결정

fsuid (filesystem uid) : 파일시스템 접근 제어 용도로 사용됨

task_struct 구조체 (8/11)

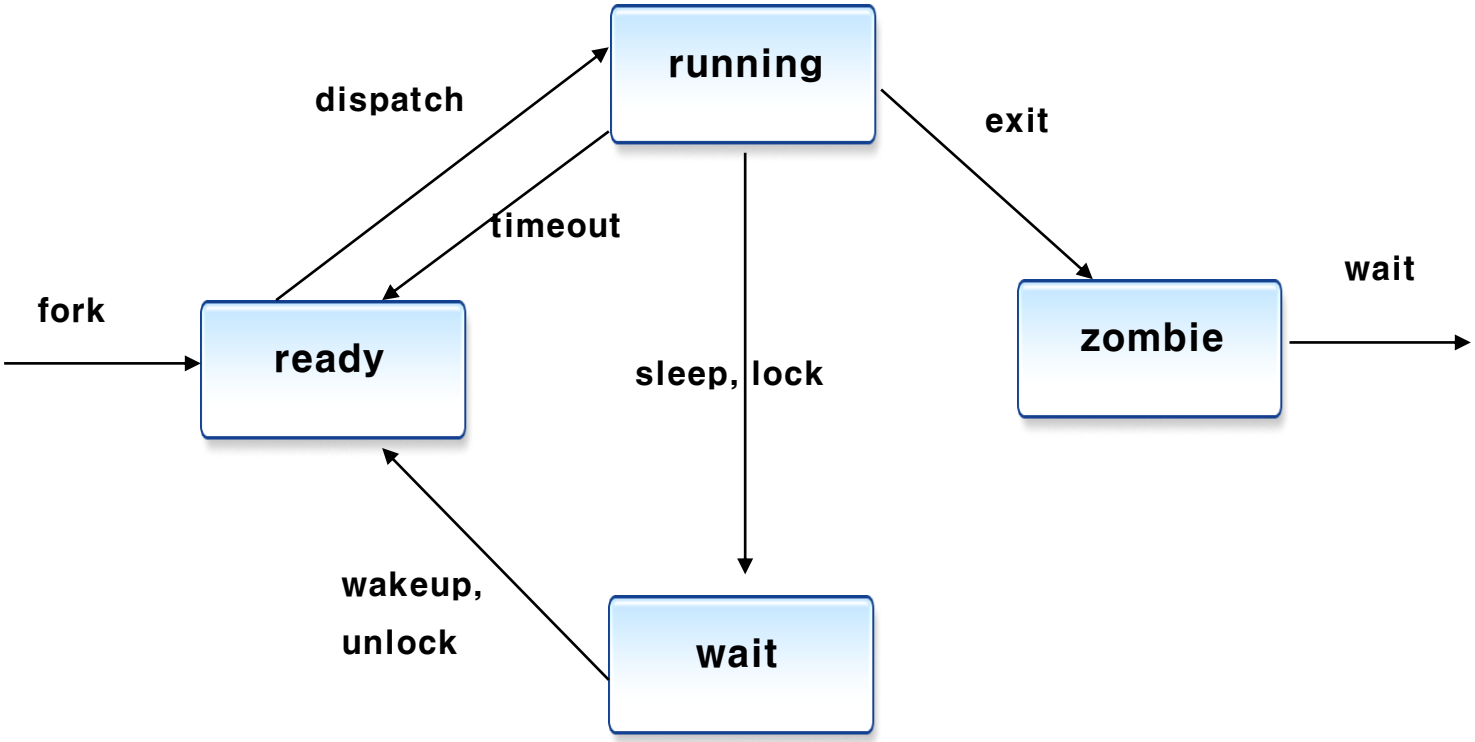
- **태스크 상태 정보에 관련된 변수**
  - state 변수

```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_STOPPED 4
#define EXIT_ZOMBIE 16
```

  - TASK_RUNNING
    - 실행 중이거나 준비 상태
  - TASK_INTERRUPT
    - 하드웨어나 시스템 자원을 사용할 수 있을 때까지 기다리고 있는 대기 상태
    - 예) wait for a semaphore
  - TASK_UNINTERRUPT
    - 하드웨어적인 조건을 기다리는 상태, 시그널을 받아도 무시
    - 예) process가 장치 파일을 열 때 해당 장치 드라이버가 자신이 다룰 하드웨어 장치가 있는지 조사할 때, memory swapping
  - TASK_STOPPED
    - 수행 중단 상태 (시그널을 받거나 트레이싱 등)
  - EXIT_ZOMBIE
    - process 실행은 종료했지만 아직 process의 자원을 반환하지 않은 상태

task_struct 구조체 (9/11)

- 태스크 상태와 전이



task_struct 구조체 (10/11)

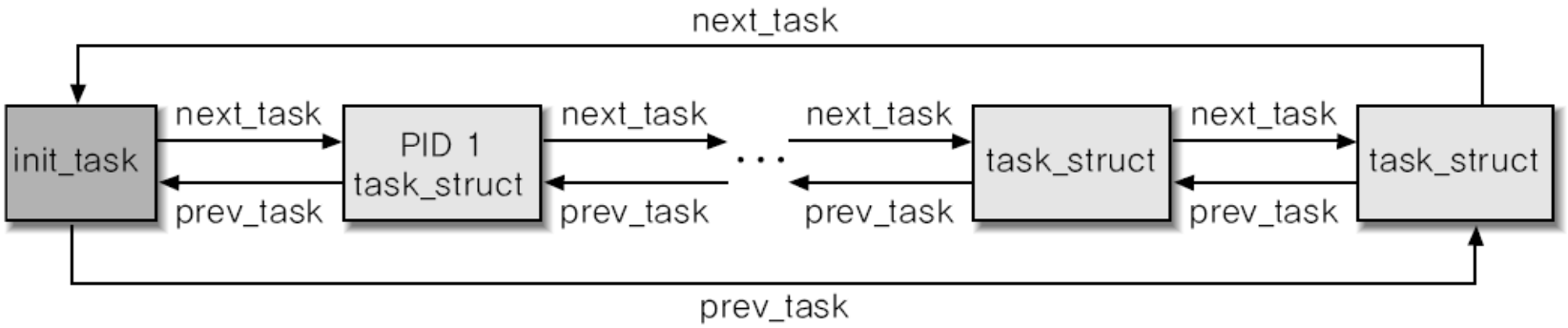
- **태스크 관계와 관련된 변수들**

```
struct task_struct *real_parent, *parent;
```

  - real_parent : 원래 부모 태스크(실제로 fork한 task)
  - parent : 현재 부모 태스크(SIGCHLD signal을 받는 task)

```
struct list_head tasks, children, sibling;
```

  - 커널에 존재하는 모든 태스크들은 원형 이중 연결 리스트로 연결



task_struct 구조체 (11/11)

- **그 밖의 변수들**
  - 스케줄링 정보
  - 시그널 정보
  - 메모리 정보
  - 파일 정보
  - 문맥 교환 정보
  - 시간 정보
  - 자원 사용 정보 등

Lab. 1 (1/3)

- **커널에 존재하는 모든 태스크들의 pid와 이름을 출력하여 확인하는 프로그램**
  - 사용할 변수

```
struct task_struct {
        ...
    pid_t pid;                  // task 식별자
        ...
    char comm[TASK_COMM_LEN]; /* executable name excluding path
                                 - access with [gs]et_task_comm (which lock
                                   it with task_lock())
        ...                      - initialized normally by setup_new_exec */
};
```

  - 사용할 매크로
    - include/linux/sched/signal.h

```
#define next_task(p) \
        list_entry_rcu((p)->tasks.next, struct task_struct, tasks)
```

Lab. 1 (2/3)

- 커널에 존재하는 모든 태스크들의 pid와 이름을 출력하여 확인하는 프로그램

displaytask.c + (~/test_task) - VIM

```
 1 #include <linux/module.h>
 2 #include <linux/sched.h>
 3 #include <linux/init_task.h>
 4
 5 int displaytask_init(void)
 6 {
 7     struct task_struct *findtask = &init_task;
 8
 9     do
10     {
11         printk("%s[%d] -> ", findtask->comm, findtask->pid);
12         findtask = next_task(findtask);
13     }
14     while( (findtask->pid != init_task.pid));
15     printk("%s[%d]\n", findtask->comm, findtask->pid);
16     return 0;
17 }
18
19 void displaytask_exit(void)
20 {
21 }
22
23 module_init(displaytask_init);
24 module_exit(displaytask_exit);
25 MODULE_LICENSE("GPL");
```

Lab. 1 (3/3)

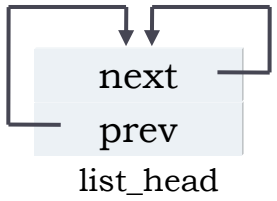
- **커널에 존재하는 모든 태스크들의 pid와 이름을 출력하여 확인하는 프로그램 (cont'd)**
  - 결과 화면

```
root@ubuntu:/home/sslab/test_task# dmesg
[ 3783.031518] init swapper/0[0]
[ 3783.031518] swapper/0[0] ->
[ 3783.031519] systemd[1] ->
[ 3783.031519] kthreadd[2] ->
[ 3783.031520] rcu_gp[3] ->
[ 3783.031520] rcu_par_gp[4] ->
[ 3783.031521] kworker/0:0H[6] ->
[ 3783.031521] mm_percpu_wq[8] ->
[ 3783.031522] ksoftirqd/0[9] ->
[ 3783.031522] rcu_sched[10] ->
[ 3783.031523] rcu_bh[11] ->
[ 3783.031523] migration/0[12] ->
[ 3783.031524] cpuhp/0[14] ->
[ 3783.031524] cpuhp/1[15] ->
[ 3783.031525] migration/1[16] ->
[ 3783.031525] ksoftirqd/1[17] ->
[ 3783.031526] kworker/1:0H[19] ->
[ 3783.031526] cpuhp/2[20] ->
[ 3783.031526] migration/2[21] ->
[ 3783.031527] ksoftirqd/2[22] ->
[ 3783.031527] kworker/2:0H[24] ->
```

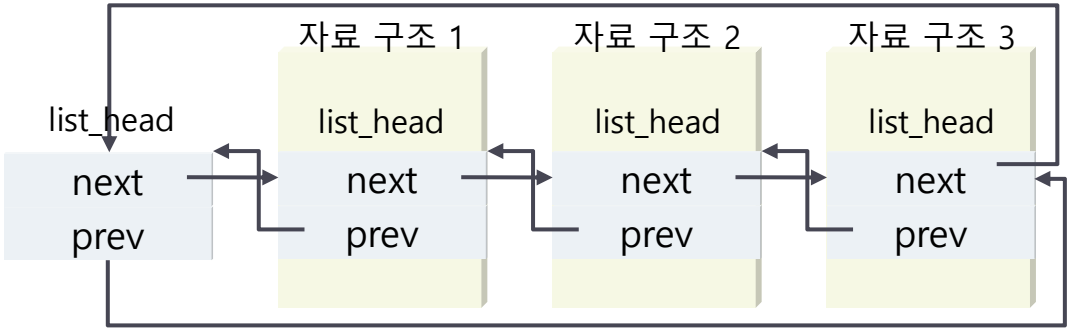
프로세스 계보 (family) (1/5)

- **이중 연결 리스트 (doubly linked list)**

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



(a) 비어있는 이중 연결 리스트



(b) 몇 개의 자료 구조를 가진 이중 연결 리스트

프로세스 계보 (family) (2/5)

- **리스트를 처리하는 매크로 및 함수**
  - list_add(n, p)
    - p 바로 다음에 n을 삽입.
  - list_add_tail(n, p)
    - p 바로 앞에 n을 삽입.
  - list_del(p)
    - p를 삭제.
  - list_empty(p)
    - 헤드가 p인 리스트가 비어있는지를 검사.
  - list_entry(p, t, m)
    - 이름이 m이고 주소가 p인 list_head 필드를 포함한 t타입 자료구조의 주소를 반환.
  - list_for_each(p, h)
    - 헤드의 주소가 h로 지정된 모든 원소를 순환.
    - 각 원소의 list_head 자료 구조의 주소가 p에 반환

프로세스 계보 (family) (3/5)

- **부모 프로세스의 디스크립터를 얻으려면**
  - 부모 프로세스의 task_struct에 대한 포인터 ➔ **parent**

```
struct task_struct *my_parent = current->parent;
```

- **자식 프로세스들의 디스크립터를 얻으려면**
  - 자식 프로세스의 task_struct에 대한 포인터 ➔ **children**

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, &current->children) {
        task = list_entry(list, struct task_struct, sibling);
        /* task now points to one of current's children */
}
```

```
list_for_each_entry(task, &current->children, list) {
        /* task now points to one of current's children */
}
```

프로세스 계보 (family) (4/5)

- **조상 프로세스를 끝까지 따라가려면**
  - init 프로세스의 task_struct에 대한 포인터

```
struct task_struct *task;
for (task = current; task != &init_task; task = task->parent)
        ;
/* task now points to init */
```

- **전체 리스트에서 다음 프로세스를 얻으려면**
  - next_task(task);

```
#define next_task(p) \ list_entry_rcu((p)->tasks.next, struct task_struct, tasks)
```

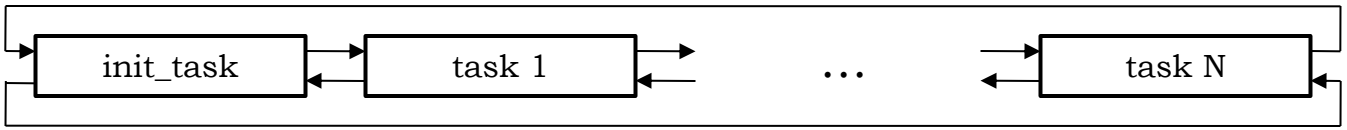
    - 이전 프로세스를 얻으려면 tasks의 prev 변수를 이용

프로세스 계보 (family) (5/5)

- **시스템 내 모든 프로세스를 출력하려면**
  - for_each_process(task)
    - 리스트에 있는 모든 프로세스를 탐색

```
struct task_struct *task;

for_each_process(task) {
        /* 각 태스크의 이름과 PID 출력 */
        printk("%s[%d]₩n", task->comm, task->pid);
}
```




```
#define for_each_process(p) ₩
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```