

# Process

# Memory

❖ 어떻게 Process가 생성될까?

## User Process

1. Fork() / pthread\_create()
2. 결국 **System Call**

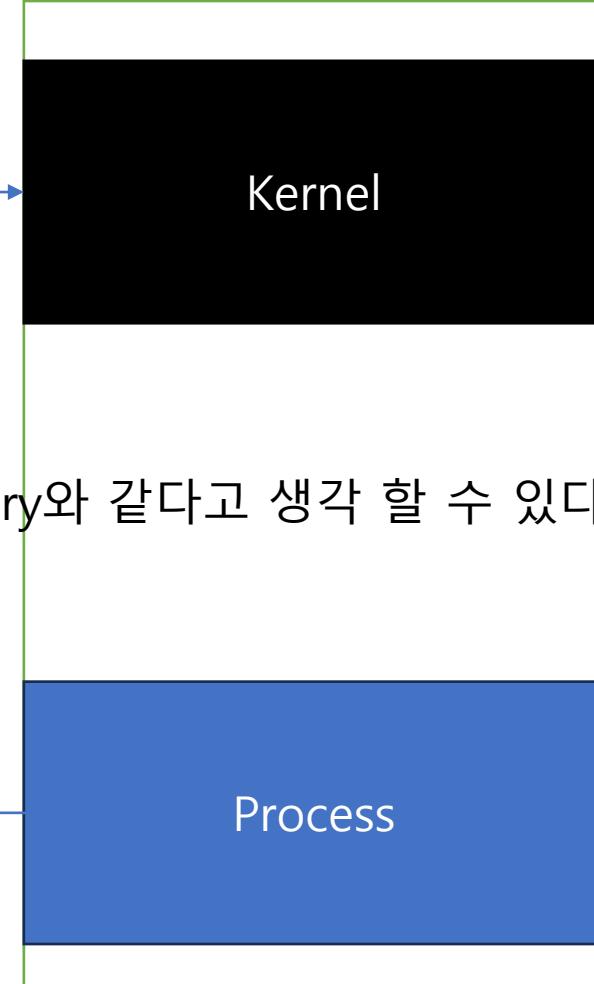
## Kernel Process

Kthread\_create() 함수 호출

결국 \_do\_fork() 호출

## System Call

※ Kernel이란 일종의 library와 같다고 생각 할 수 있다.



**init 프로세스:** 주로 부팅 과정에서 **유저 프로세스**를 생성

**kthreadd 프로세스:** 커널 레벨 프로세스(커널 스레드)를 생성

- swapper 프로세스: 0
- init 프로세스: 1
- kthreadd 프로세스: 2

unsigned long clone\_flags: 프로세스를 생성할 때 지정하는 옵션 정보를 저장

## \_do\_fork() 함수

```
long _do_fork(unsigned long clone_flags,  
              unsigned long stack_start,  
              unsigned long stack_size,  
              int __user *parent_tidptr,  
              int __user *child_tidptr,  
              unsigned long tls)
```

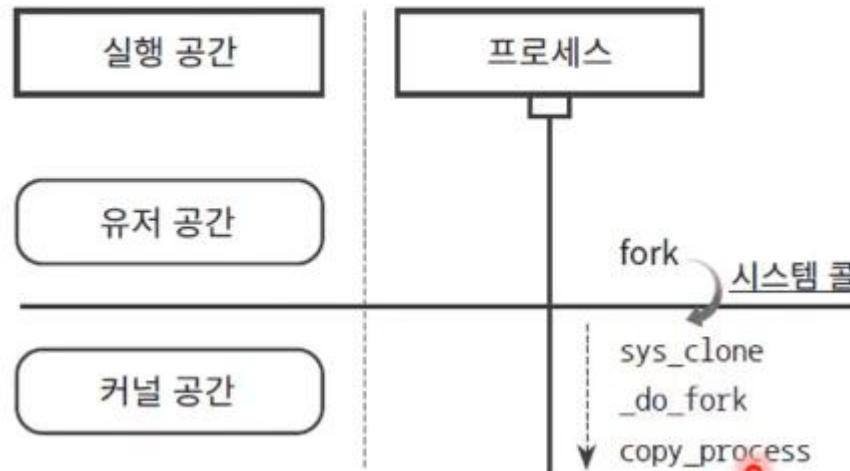
PID 반환

unsigned long stack\_size: 보통 유저 영역에서 스레드를 생성할 때 복사하려는  
스택의 주소

→ unsigned long stack\_size: 유저 영역에서 실행 중인 스택 크기

# User Process

## 유저 프로세스의 생성 흐름도



## sys\_clone() 함수 코드 분석

```
https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/fork.c
01 #ifdef __ARCH_WANT_SYS_CLONE
02 #ifdef CONFIG_CLONE_BACKWARDS
03 SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
04                  int __user *, parent_tidptr,
05                  unsigned long, tls,
06                  int __user *, child_tidptr)
07#endif
08 {
09     /* _do_fork() 함수를 그대로 호출 */
10     return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
11 }
```

Sys\_clone() → \_do\_fork()

- 유저 공간에서 fork() 함수를 호출하면 리눅스에서 제공하는 라이브러리의 도움을 받아 커널에게 프로세스 생성 요청
- 리눅스에서 제공하는 라이브러리 코드가 실행되면서 시스템 콜을 발생
- 리눅스 커널 계층에서는 fork() 함수에 대응하는 시스템 콜 핸들러인 sys\_clone() 함수를 호출

# User Process

## fork()를 통한 User process 생성 :: ftrace log 분석

01번째 로그를 출력하는 주인공은 pid가 946인 bash 프로세스

```
01 bash-946 [003] .... 676.984916: copy_process.part.5+0x14/0x1acc <-
do_fork+0xc0/0x41c
02 bash-946 [003] .... 676.984929: <stack trace>
03 => copy_process.part.5+0x18/0x1acc
04 -> _do_fork+0xc0/0x41c
05 => sys_clone+0x30/0x38
06 => ret_fast_syscall+0x0/0x28
07 -> 0x7ef11270
08 ...
09 bash-946 [003] .... 676.985385: sched_process_fork: comm=bash pid=946
child_comm=bash child_pid=1360
```

유저 공간에서 fork() 함수를 호출해 해당 시스템 콜 핸들러 함수인  
sys\_clone() 함수가 호출

```
10 raspbian_proc-1360 [001] d.h. 676.989734: sched_wakeup: comm=lxterminal pid=847
prior=120 target_cpu=001
11 raspbian_proc-1360 [001] d... 676.989763: sched_switch: prev_comm=raspbian_proc
prev_pid=1360 prev_prio=120 prev_state=S
...
12 <idle>-0 [001] dnh. 679.989823: sched_wakeup: comm=raspbian_proc pid=1360 prio=120
target_cpu=001
13 raspbian_proc-1360 [001] d... 679.989938: sched_switch: prev_comm=raspbian_proc
prev_pid=1360 prev_prio=120 prev_state=S ==> next_comm=kworker/1:3 next_pid=431
next_prio=120
...
```

11, 13, 15, 17번째 줄을 보면 raspbian\_proc 프로세스가 실행하다가 스케줄링으  
로 휴면상태로 진입

실행 시간을 보면 3초 간격으로 휴면 상태로 진입했다가 실행을 반복

※User Code에서 3초 간격 sleep() 호출

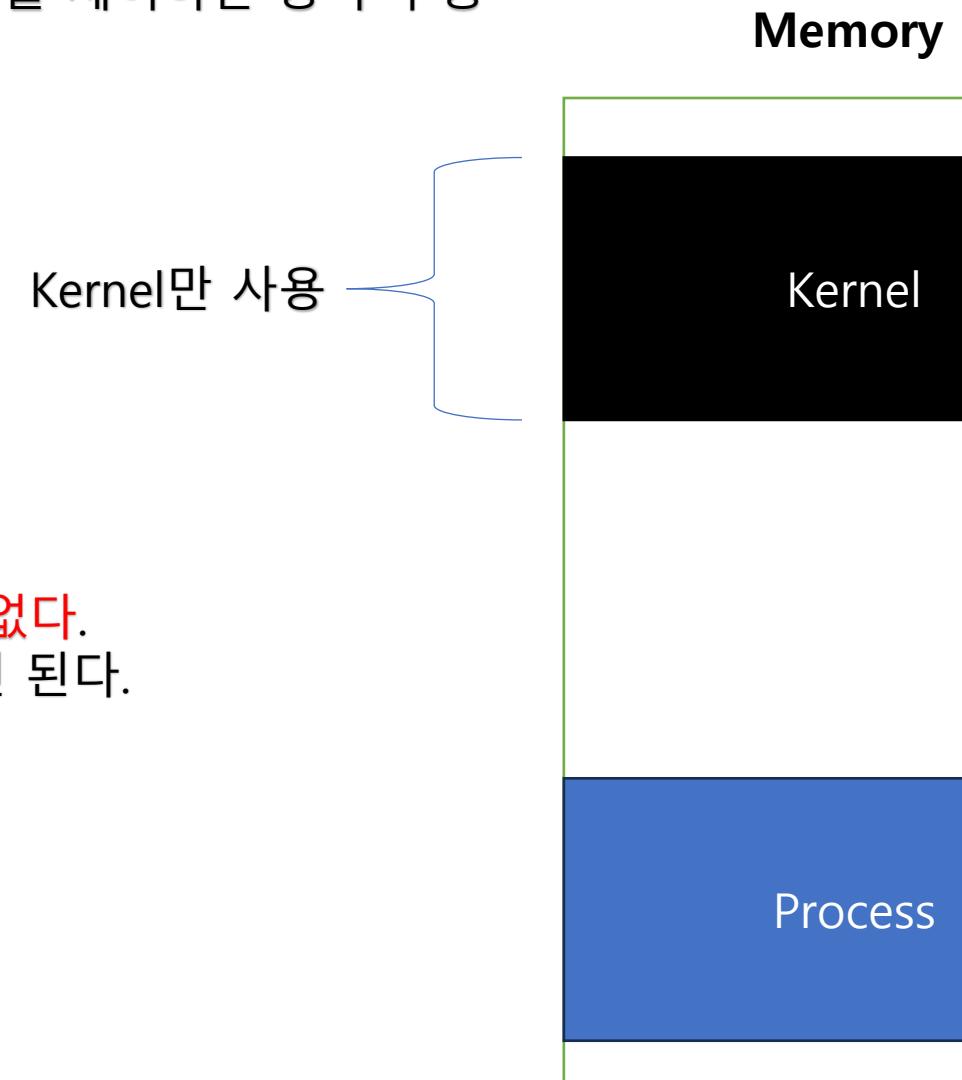
# Kernel Process

## Kernel Process (Thread)

- 커널 공간에서만 실행되는 프로세스
- 백그라운드 작업으로 실행 되면서 시스템 메모리나 전원을 제어하는 동작 수행

## Kernel Process 특징

- 실행 및 Sleep을 커널에서 직접 제어
- 대부분 시스템과 생명주기가 동일하다.



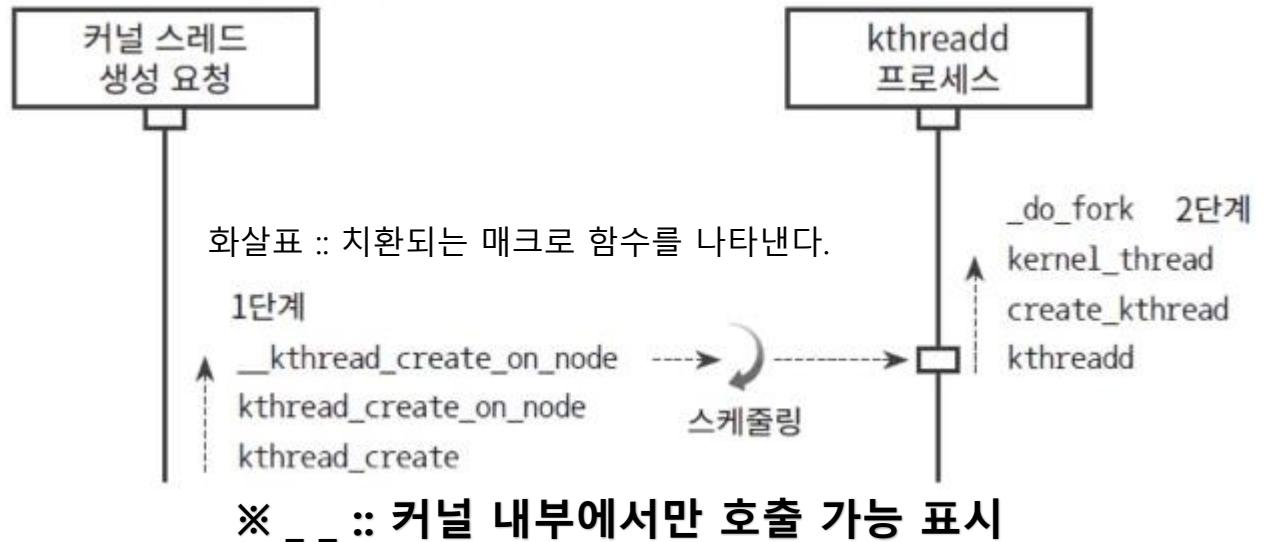
## 내 이해 정리

- 생각해보면, kernel Process는 System Call을 할 필요가 없다.
- 그냥 본인이 kernel code 사용해서 Sleep 및 실행을 하면 된다.
- 왜냐하면 **에초에 kernel에서 수행되기 때문이다.**

# Kernel Thread 종류

- kthreadd 프로세스
  - kthreadd 프로세스는 모든 커널 스레드의 부모 프로세스
  - 스레드 핸들러 함수는 kthreadd()이며, 커널 스레드를 생성하는 역할을 수행
- 워커 스레드
  - 워크큐에 큐잉된 워크(Work)를 실행하는 프로세스
  - 스레드 핸들러 함수는 worker\_thread()이며, process\_one\_work() 함수를 호출해 워크를 실행
- ksoftirqd 프로세스
  - Soft IRQ를 위해 실행되는 프로세스
  - ksoftirqd 스레드의 핸들러는 run\_ksoftirqd() 함수로서 Soft IRQ 서비스를 실행
  - Soft IRQ 서비스를 처리하는 \_do\_softirq() 함수에서 ksoftirqd를 깨움
- IRQ 스레드( threaded IRQ)
  - 인터럽트 후반부 처리를 위해 쓰이는 프로세스

# Kernel Process 생성



<https://github.com/raspberrypi/linux/blob/rpi-4.19.y/drivers/vhost/vhost.c>

```
1 long vhost_dev_set_owner(struct vhost_dev *dev)
2 {
3     struct task_struct *worker;
4     int err;
...
5     /* No owner, become one */
6     dev->mm = get_task_mm(current);
7     worker = kthread_create(vhost_worker, dev, "vhost-%d", current->pid);
```

- kthread\_create() 함수의 첫 번째 인자로 vhost\_worker로 스레드 핸들러 함수의 이름을 지정
- vhost\_dev 구조체의 주소를 2번째 인자로 전달
- 3번째 인자로 "vhost-%d"를 전달하며 커널 스레드의 이름을 나타냄

- 1단계: kthreadd 프로세스에게 커널 프로세스 생성을 요청
- 2단계: kthreadd 프로세스는 깨어나 프로세스를 생성해달라는 요청이 있으면 프로세스를 생성

# Kernel Process 생성

- 1단계: kthreadd 프로세스에게 커널 스레드 생성을 요청
  - kthread\_create() 함수
  - kthread\_create\_on\_node() 함수
  - \_\_kthread\_create\_on\_node() 함수

## • kthread\_create() 함수 분석

```
https://github.com/raspberrypi/linux/blob/rpi-4.19.y/include/linux/kthread.h
1 #define kthread_create(threadfn, data, namefmt, arg...) \
2     kthread_create_on_node(threadfn, data, NUMA_NO_NODE, namefmt, ##arg)
3
4 struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
5                                             void *data, int node,
6                                             const char namefmt[],
7                                             ...)
```

### • 함수 인자

- **int (\*threadfn)(void \*data)**: 스레드 핸들러 함수 주소를 저장하는 필드
- **void \*data**: 스레드 핸들러 함수로 전달하는 매개변수
- **int node**: 노드 정보
- **const char namefmt[]**: 커널 스레드 이름을 저장

### • 코드 분석:

- 커널 컴파일 과정에서 전처리기는 kthread\_create() 함수를 kthread\_create\_on\_node() 함수로 교체
- 커널이나 드라이버 코드에서 kthread\_create() 함수를 호출하면 실제로 동작하는 코드는 kthread\_create\_on\_node() 함수
- kthread\_create\_on\_node() 함수 내부에서 \_\_kthread\_create\_on\_node() 함수 호출

# Kernel Process 생성

- 1단계: kthreadd 프로세스에게 커널 스레드 생성을 요청
  - kthread create() 함수
  - kthread\_create\_on\_node() 함수
  - \_\_kthread\_create\_on\_node() 함수

## • kthread\_create\_on\_node() 함수 분석

```
https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/kthread.c#L270
01 struct task_struct *kthread_create_on_node(int (*threadfn) (void *data),
02                                              void *data, int node,
03                                              const char namefmt[],
04                                              ...)
05 {
06     struct task_struct *task;
07     va_list args;
08
09     va_start(args, namefmt);
10     task = __kthread_create_on_node(threadfn, data, node, namefmt, args); 호출하는 역할을 한다.
11     va_end(args);
12
13     return task;
14 }
```

# Kernel Process 생성

- 1단계: kthreadd 프로세스에게 커널 스레드 생성을 요청
  - kthread\_create() 함수
  - kthread\_create\_on\_node() 함수
  - \_\_kthread\_create\_on\_node() 함수

## • \_\_kthread\_create\_on\_node() 함수 분석

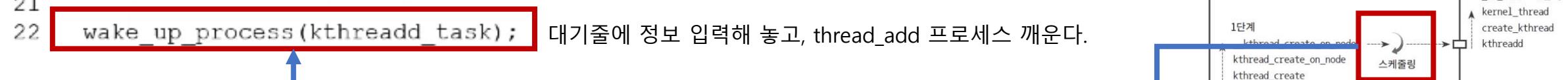
<https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/kthread.c#L270>

```
1 struct task_struct *__kthread_create_on_node(int (*threadfn)(void *data),  
2                                              void *data, int node,  
3                                              const char namefmt[],  
4                                              va_list args)  
5 {
```

```
6     DECLARE_COMPLETION_ONSTACK(done);  
7     struct task_struct *task;           Task 자료구조  
8     struct kthread_create_info *create = kmalloc(sizeof(*create),  
9                                               GFP_KERNEL);  
10    Kernel Thread 의 생성과정을 관리하는 주요 자료구조이다.  
11    해당 자료구조 생성을 위해 메모리 할당.  
12    if (!create)  
13        return ERR_PTR(-ENOMEM); 위에서 만든 자료구조에 값 할당
```

```
13     create->threadfn = threadfn; 커널 쓰레드 함수의 시작 주소  
14     create->data = data;  
15     create->node = node;  
16     create->done = &done;
```

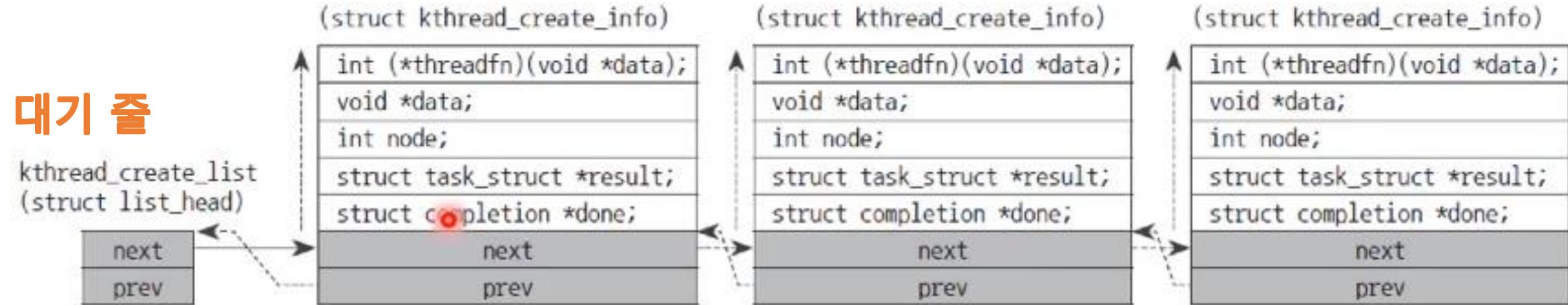
```
17     spin_lock(&kthread_create_lock);  
18     list_add_tail(&create->list, &kthread_create_list);  
19     spin_unlock(&kthread_create_lock);  
20  
21     wake_up_process(kthreadd_task); 대기줄에 정보 입력해 놓고, thread_add 프로세스 깨운다.
```



# Kernel Process 생성

- 1단계: kthreadd 프로세스에게 커널 스레드 생성을 요청
  - kthread\_create() 함수
  - kthread\_create\_on\_node() 함수
  - \_kthread\_create\_on\_node() 함수

- kthreadd() 함수에서 커널 스레드를 생성할 때 자료구조



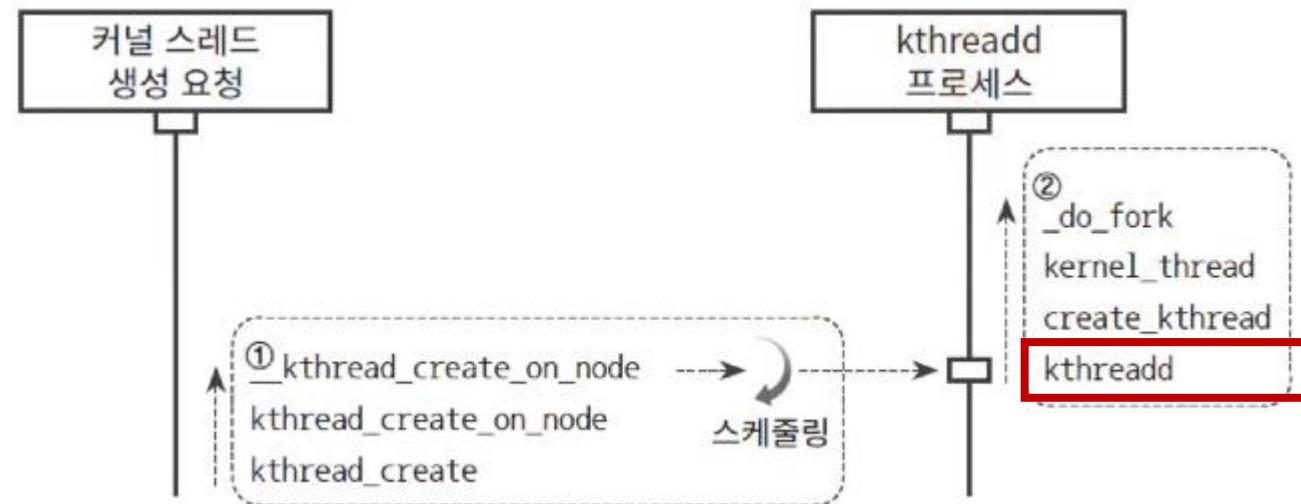
**대기 정보 :: 이 정보를 토대로 Kernel Thread 생성 요청**

# Kernel Process 생성 – Kthread add()

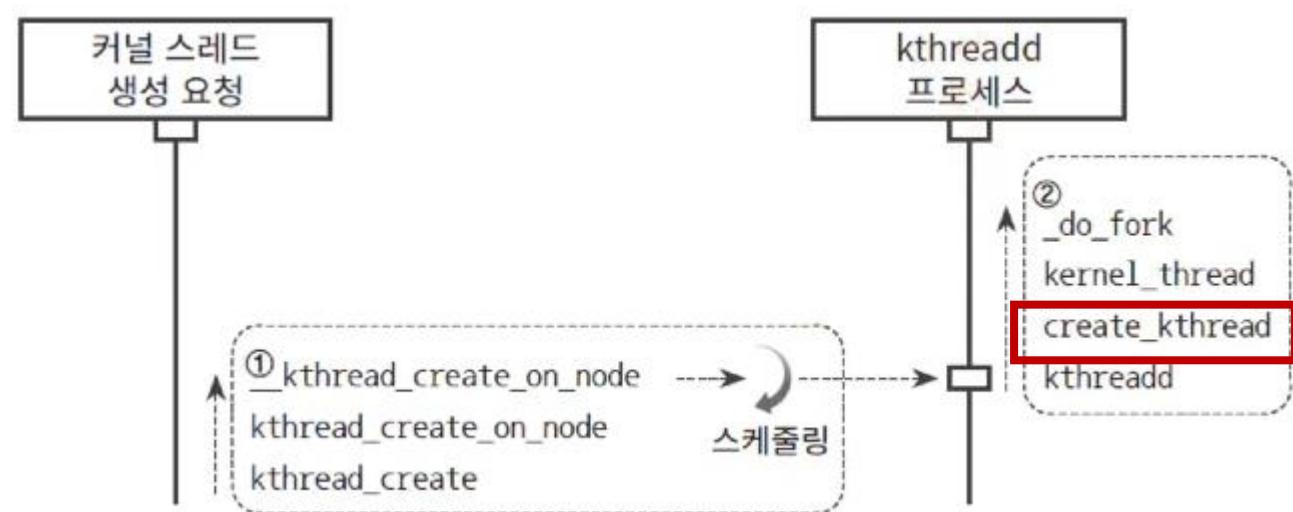
☞ 대기 줄 확인 후, thread 생성함수 호출

## ● kthreadd() 함수 분석

```
https://github.com/raspberrypi/linux/blob/rpi-4.19.y/ker  
1 int kthreadd(void *unused)  
2 {  
...  
14     for (;;) {  
15         set_current_state(TASK_INTERRUPTIBLE);  
16         if (list_empty(&kthread_create_list))  
17             schedule(); /* schedule() 함수를 호출해 스스로 휴면 상태로 진입 */  
18         set_current_state(TASK_RUNNING); 깨어나면, 여기서부터 시작한다.  
19  
20         spin_lock(&kthread_create_lock);  
/* kthread_create_list 연결 리스트가 비어있지 않으면 21~32번째 줄을 실행해 커널 스레드를 생성 */  
21         while (!list_empty(&kthread_create_list)) {  
22             struct kthread_create_info *create;  
23             /* kthread_create_list.next 필드를 통해 kthread_create_info 구조체의 주소를 읽음 */  
24             create = list_entry(kthread_create_list.next,  
25                                 struct kthread_create_info, list);  
26             list_del_init(&create->list);  
27             spin_unlock(&kthread_create_lock);  
28             /* create_kthread() 함수를 호출해서 커널 스레드를 생성 */  
29             create_kthread(create);  
...  
32         }  
33         spin_unlock(&kthread_create_lock);  
34     }  
}
```



# Kernel Process 생성 – create\_kthread()



## • create\_kthread() 함수 분석

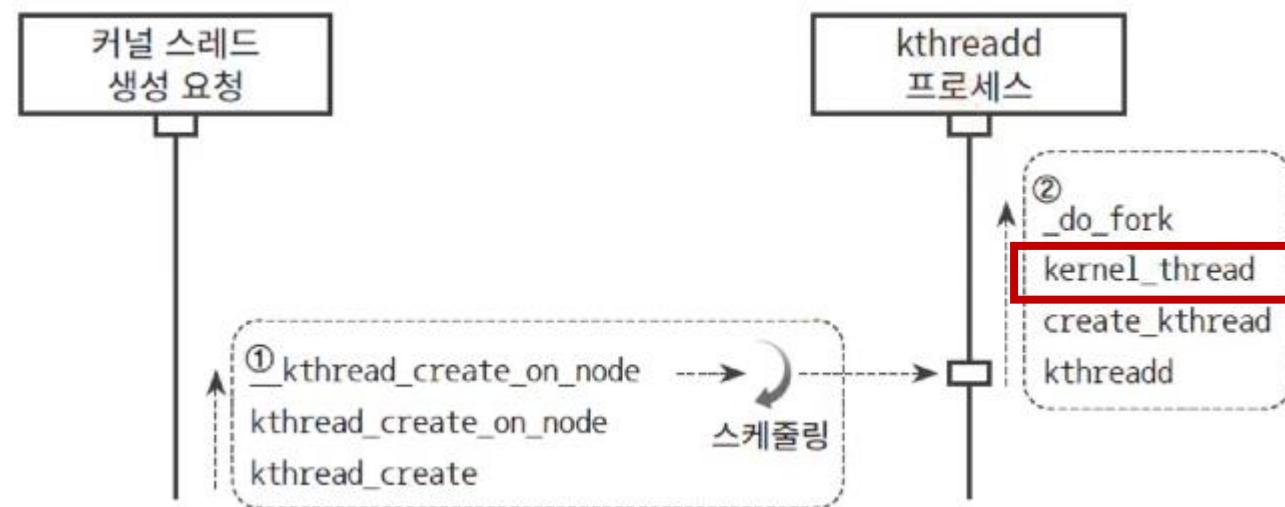
<https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/kthread.c>

```
1 static void create_kthread(struct kthread_create_info *create)
2 {
3     int pid;
4
5 #ifdef CONFIG_NUMA
6     current->pref_node_fork = create->node;
7 #endif
8     /* We want our own signal handler (we take no signals by default). */
9     pid = kernel_thread(kthread, create, CLONE_FS | CLONE_FILES | SIGCHLD);
```

## • 코드 분석:

- 9번째 줄과 같이 kernel\_thread() 함수를 호출
- CLONE\_FS, CLONE\_FILES, SIGCHLD 매크로들 OR 연산한 결과를 3번째 인자로 설정

# Kernel Process 생성 – create\_kthread()



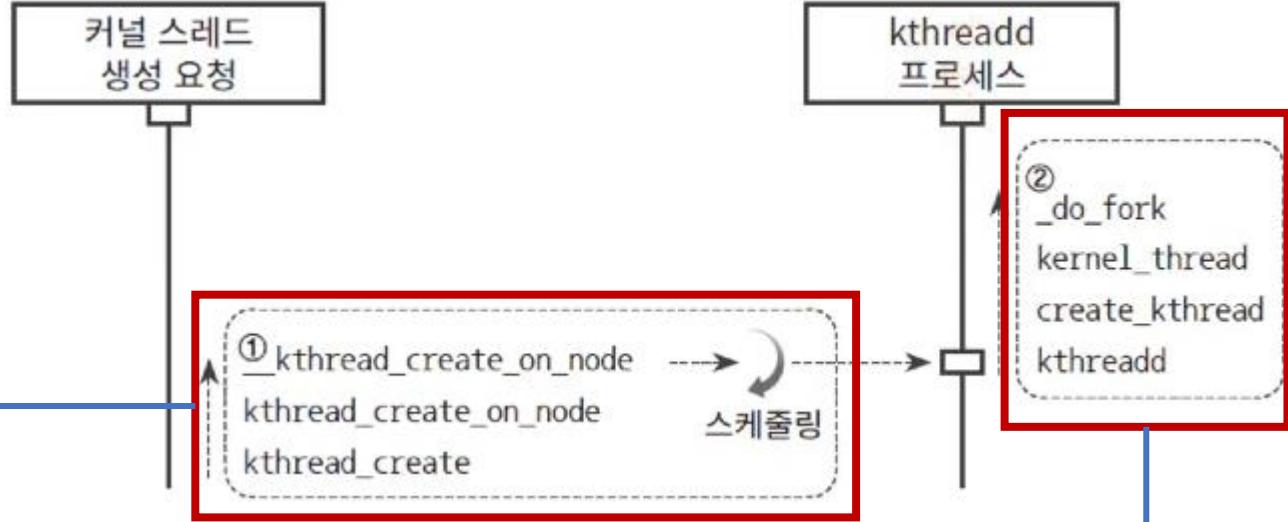
## ● kernel\_thread() 함수 분석

```
/*
 * Create a kernel thread.
 */
01 pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
02 {
03     return _do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn,
04                      (unsigned long)arg, NULL, NULL, 0);
05 }
```

## ● 코드 분석:

- 03~04번째 줄과 같이 \_do\_fork() 함수 호출
- CLONE\_FS, CLONE\_FILES, SIGCHLD 매크로들 OR 연산한 결과를 3번째 인자로 설정

# Kernel Process 생성 – 정리



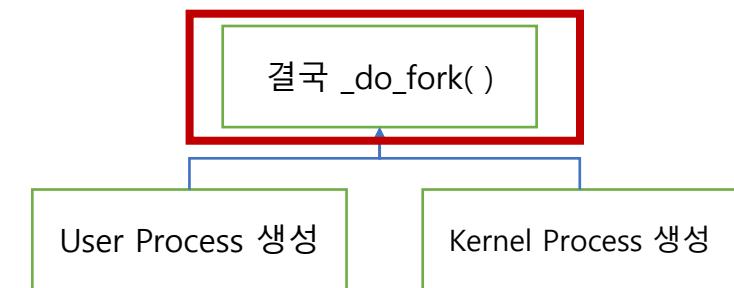
- Q: 커널 스레드를 생성하려면 어떤 함수를 호출해야 할까?
  - A: `kthread_create()` 함수를 호출해야 하며 함수의 인자로 '스레드 핸들러 함수', '매개변수', '커널 스레드 이름'을 지정
- Q: 커널 스레드를 생성하는 단계는 무엇일까?
  - A: kthreadd 프로세스에게 커널 스레드 생성을 요청한 후 kthreadd 프로세스를 깨움. kthreadd 프로세스는 깨어나 자신에게 커널 스레드 생성 요청이 있었는지 확인한 후 만약 생성 요청이 있다면 커널 스레드를 생성

# Process 생성

## ☞ \_do\_fork() 실행 흐름

### ● \_do\_fork() 함수 분석

```
https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/fork.c
01 long _do_fork(unsigned long clone_flags, unsigned long stack_start,
03             unsigned long stack_size, int __user *parent_tidptr,
05             int __user *child_tidptr, unsigned long tls)
07 {
...
/* 부모 프로세스의 메모리 및 시스템 정보를 자식 프로세스에게 복사*/
14     p = copy_process(clone_flags, stack_start, stack_size,
15                     child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
16     add_latent_entropy();
17
18     if (IS_ERR(p)) /* p라는 포인터 형 변수에 오류가 있는지 검사 */
19         return PTR_ERR(p); /* 오류 코드를 반환하면서 함수의 실행을 종료*/
20
21     trace_sched_process_fork(current, p); /*sched_process_fork 이벤트에 대한 메시지 출력*/
22
23     pid = get_task_pid(p, PIDTYPE_PID); /* 22~23번째 줄: pid는 계산해 nr 지역변수에 저장*/
24     nr = pid_vnr(pid);
...
25     wake_up_new_task(p); /* 생성한 프로세스를 깨운 */
...
26     put_pid(pid);
27     return nr; /* 프로세스의 PID를 담고 있는 정수형 타입의 nr 지역변수를 반환*/
28 }
```



### STEP 1 Copy\_process 호출

- 부모Process 리소스 복사를 통해  
프로세스 생성

### STEP 2

생성한 프로세스를 깨운다.

# Process 생성

## ☞ copy\_process() 실행 흐름

### ● copy\_process() 함수 분석[1/2]

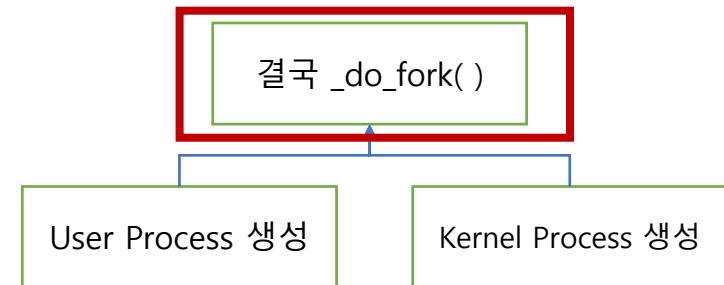
```
https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/fork.c
1 static __latent_entropy struct task_struct *copy_process(
2                                     unsigned long clone_flags, unsigned long stack_start,
3                                     unsigned long stack_size, int __user *child_tidptr,
4                                     struct pid *pid, int trace,
5                                     unsigned long tls, int node)
6
7                                     {
8
9                                     int retval;
10
11                                     struct task_struct *p;    Task_struct == Process의 자료구조 (PCB)
12
13                                     retval = -ENOMEM;
```

```
/* 14번째 줄: 생성할 프로세스의 태스크 디스크립터인 task_struct 구조체와 프로세스가
15 실행될 스택 공간을 할당. dup_task_struct() 함수를 호출해 태스크 디스크립터를 p에 저장 */
16                                     p = dup_task_struct(current, node);
```

```
17
18                                     if (!p)
19                                         goto fork_out;
...
20                                     /* Perform scheduler related setup. Assign this task to a CPU. */
```

```
/* 태스크 디스크립터를 나타내는 task_struct 구조체에서 스케줄링 관련 정보를 초기화 */
21                                     retval = sched_fork(clone_flags, p);
22
23                                     if (retval)
24                                         goto bad_fork_cleanup_policy;
```

```
/* Continued... */
```



User Process 생성

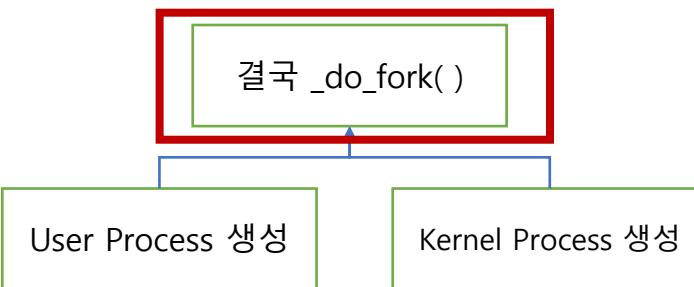
Kernel Process 생성

결국 \_do\_fork()

여기서 memor가 할당된다는 건가..?  
일단 PCB가 생성되는건 맞는 것 같고  
Stack이 할당되는건 메모리 할당이 맞는 것 같다.

# Process 생성

## ☞ copy\_process() 실행 흐름



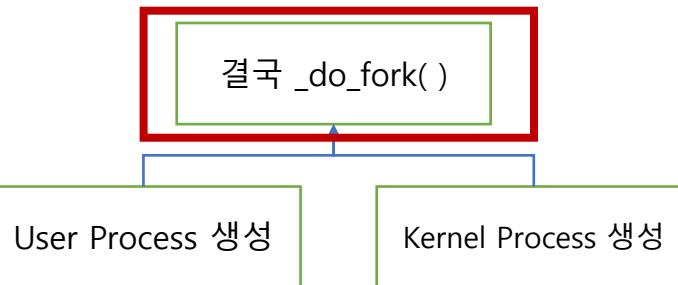
### ● copy\_process() 함수 분석[2/2]

<https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/fork.c>

```
1 static __latent_entropy struct task_struct *copy_process(
2             unsigned long clone_flags, unsigned long stack_start,
4             unsigned long stack_size, int __user *child_tidptr,
6             struct pid *pid, int trace,
8             unsigned long tls, int node)
10 {
...
...
/* 22~27번째 줄: 프로세스의 파일 디스크립터 관련 내용(파일 디스크립터, 파일 디스크립터 테이블)을 초기화 */
22     retval = copy_files(clone_flags, p);
23     if (retval)
24         goto bad_fork_cleanup_semundo;
25     retval = copy_fs(clone_flags, p);
26     if (retval)
27         goto bad_fork_cleanup_files;
...
/* 프로세스가 등록한 시그널 핸들러 정보인 sighand_struct 구조체를 생성해서 복사 */
28     retval = copy_sighand(clone_flags, p);
29     if (retval)
30         goto bad_fork_cleanup_fs;
```

# Process 생성

## ☞ wake\_up\_new\_task( ) 실행 흐름



### ● wake\_up\_new\_task() 함수 분석

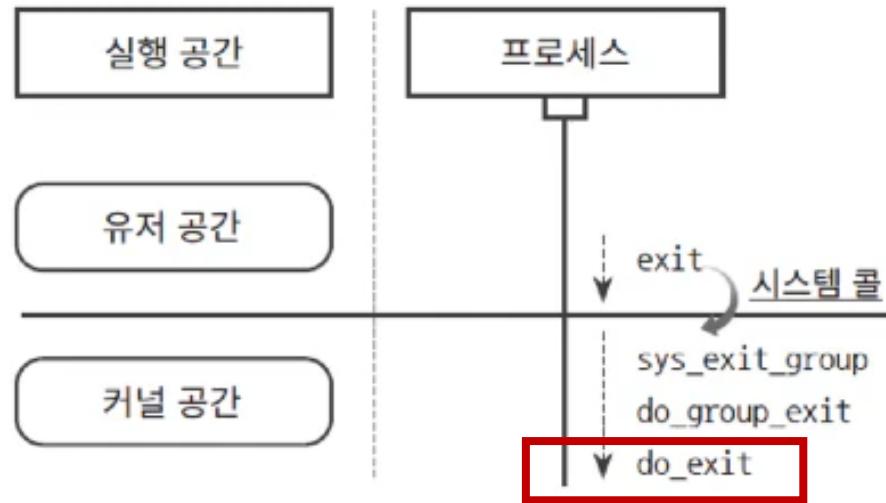
<https://github.com/raspberrypi/linux/blob/rpi-4.19.y/kernel/sched/core.c>

```
1 void wake_up_new_task(struct task_struct *p)
2 {
3     struct rq_flags rf;
4     struct rq *rq;
5
6     add_new_task_to_grp(p);
7     raw_spin_lock_irqsave(&p->pi_lock, rf.flags);
8
9     p->state = TASK_RUNNING; /* 프로세스 상태를 TASK_RUNNING으로 바꿈 */
10 #ifdef CONFIG_SMP
/* 11번째 줄: 프로세스의 thread_info 구조체의 cpu 필드에 현재 실행 중인 CPU 번호를 저장 */
11     set_task_cpu(p, select_task_rq(p, task_cpu(p), SD_BALANCE_FORK, 0, 1));
12 #endif
13     rq = __task_rq_lock(p, &rf); /* 런큐 주소를 읽음 */
14     update_rq_clock(rq);
15     post_init_entity_util_avg(&p->se);
16
17     mark_task_starting(p);
/* activate_task() 함수를 호출해 런큐에 새롭게 생성한 프로세스를 삽입*/
18     activate_task(rq, p, ENQUEUE_NOCLOCK);
```

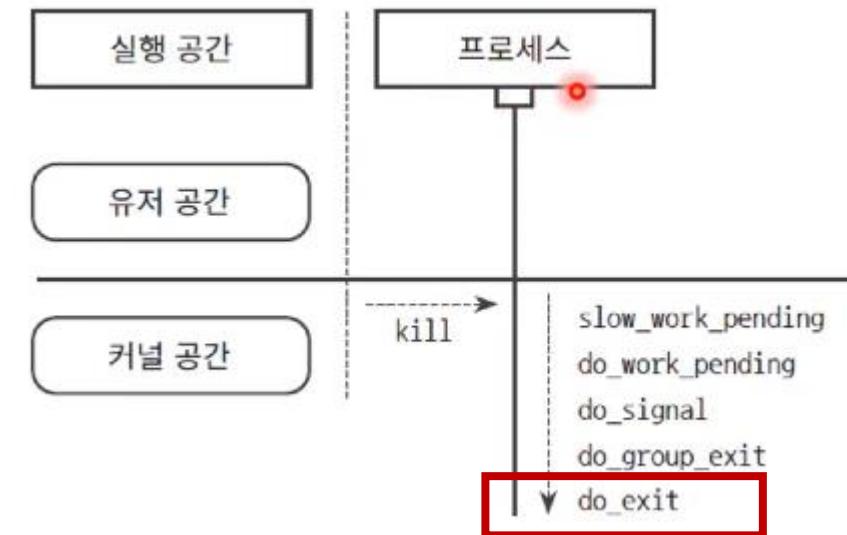
# Process 종료

## Process 종료의 2가지 방법

- ① user Process가 exit() 호출



- ② Kill 시그널을 받아 process 종료



User Process는 언제 kill 시그널을 받는가?

1. Interrupt handler 처리 후 User Mode 복귀 전
2. System Call 종료 후 User Mode 복귀 전



1. 리소스 해제
2. 부모 프로세스에 종료 알림
3. Task\_struct의 state = TASK\_DEAD
4. Do\_task\_dead( ) 호출

☞ 왜 Schedule() 함수 호출?

- Process는 자신의 프로세스 스택 메모리 공간을 해제 할 수 없다.
- 따라서, schedule() 함수를 호출해 스케줄링한 후 **다음에 실행 되는 프로세스가 종료되는 프로세스의 메모리 공간을 해제**

/\* 이전에 실행했던 프로세스 상태가 TASK\_DEAD일 때 05~12번째 줄을 실행하는 조건문 \*/

```

04     if (unlikely(prev_state == TASK_DEAD)) {
05         if (prev->sched_class->task_dead)
06             prev->sched_class->task_dead(prev);
07
08         kprobe_flush_task(prev);
09

```

**다른 프로세스가 PCB & Stack 해제**

/\* put\_task\_stack() 함수를 호출해서 프로세스의 스택 메모리 공간을 해제하고 키널 메모리 공간에 반환

```

*/●
10         put_task_stack(prev);
11

```

/\* put\_task\_struct() 함수를 실행해 프로세스를 표현하는 자료구조인 task\_struct가 위치한 메모리를 해제

```

12         put_task_struct(prev);
13     }
  
```