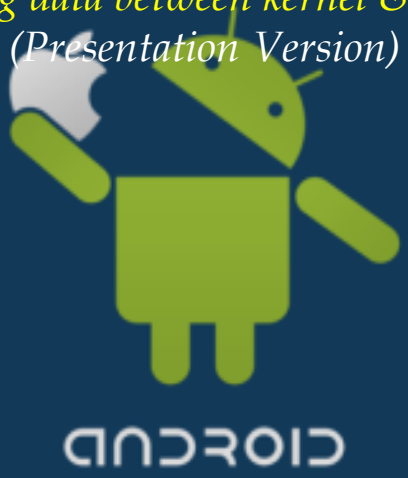
Android Device Driver Hacks

: interrupt handler, tasklet, work queue, kernel thread, synchronization, transferring data between kernel & user space

(Presentation Version)



chunghan.yi@gmail.com, slowboot

목차*

(*) 본 문서가 아래의 질문에 대해 적절한 답을 주고 있는가 ?*

1) Task scheduling과 kernel preemption의 개념이 잘 설명되어 있는가 ?
2) Interrupt context와 process context란 무엇인가 ?
3) Interrupt context(interrupt handler 등)에서 해서는 안되는 일이 무엇인가 ?
4) Process context에서 주의해야 할 사항은 무엇인가 ?
5) Top half와 bottom half의 개념이 잘 설명되어 있는가 ?
6) Shared IRQ의 개념이 잘 설명되어 있는가 ?
7) Interrupt를 disable해야 하는 이유와 방법이 잘 설명되어 있는가 ?
8) Tasklet은 언제 사용하는가 ?
9) Work queue는 언제 사용하는가 ?
10) Kernel thread는 언제 사용하는가 ?
11) Threaded interrupt handler는 언제 사용하며, 주의할 사항은 무엇인가 ?
12) Concurrency 상황이 언제이며, 이때 어찌(어떻게 programming)해야 하는가 ?
13) Top/bottom half, interrupt/process context, SMP등 각각의 상황 별 locking 방법이 적절히 기술되어 있는가 ?
14) Timer를 사용하려면 어찌해야 하는가 ?
15) Kernel과 user process가 통신하려면 어찌해야 하는가 ?
16) wakelock, suspend/resume의 개념이 잘 설명되어 있는가 ?
17) Platform driver를 작성하려면 어찌해야 하는가 ?

(*) Keywords

- *Task*
- *Preemptive kernel*
- *Scheduling(run queue, wait queue, priority)*
- *Interrupt context & process context, context switching*
- *Top half, Bottom half*
- *Interrupt handler*
- *Tasklet, Work queue, Kernel Thread, Threaded Interrupt handler*

- *Timer & Hrtimer*

- *Concurrency, critical region(section)*
- *Synchronization, locking*

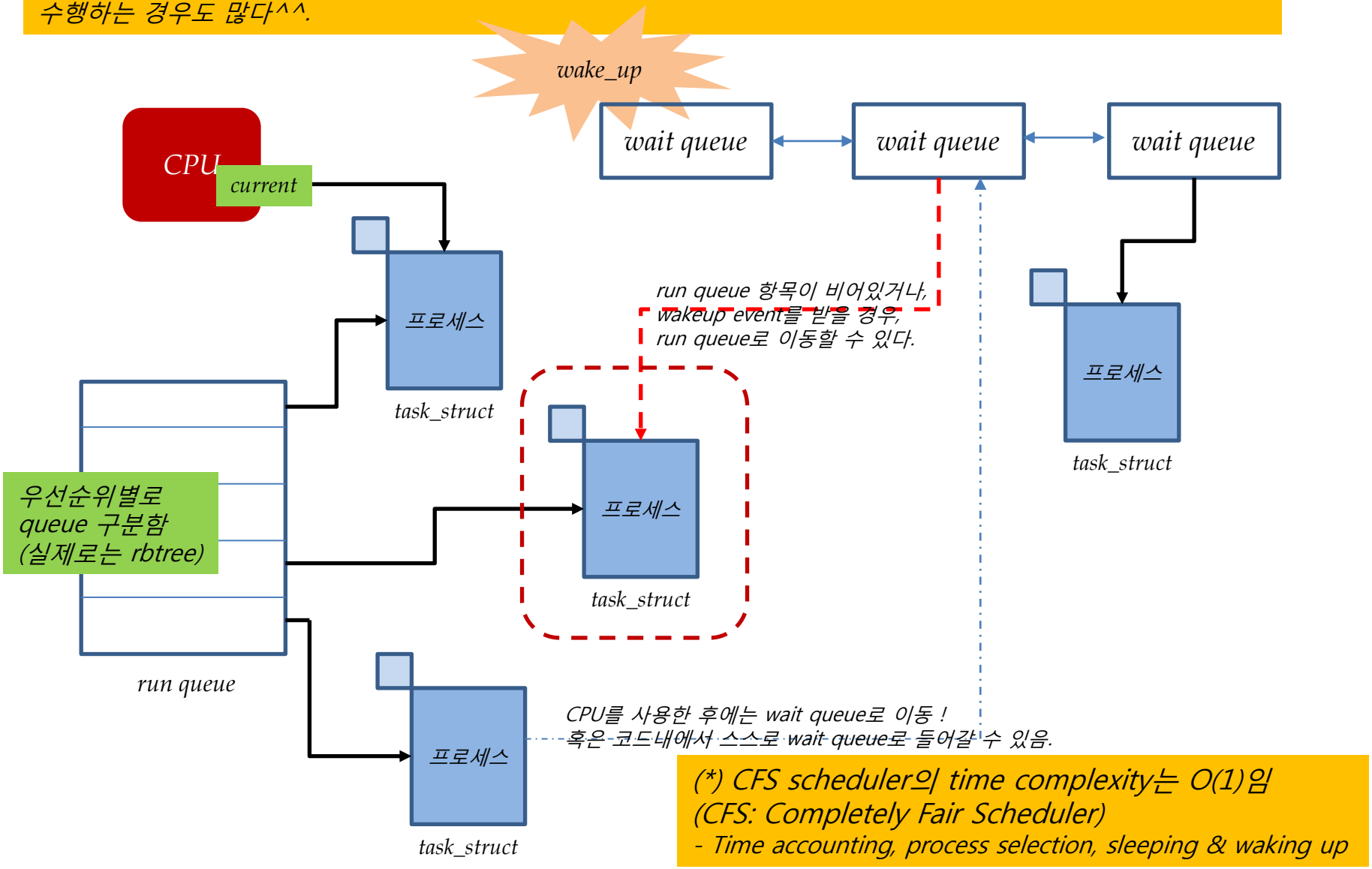
- *procfs, sysfs*
- *mmap*
- *ashmem, binder* ← *android*

- *Resume/suspend/wakelock* ← *android*

- *Platform device & driver*

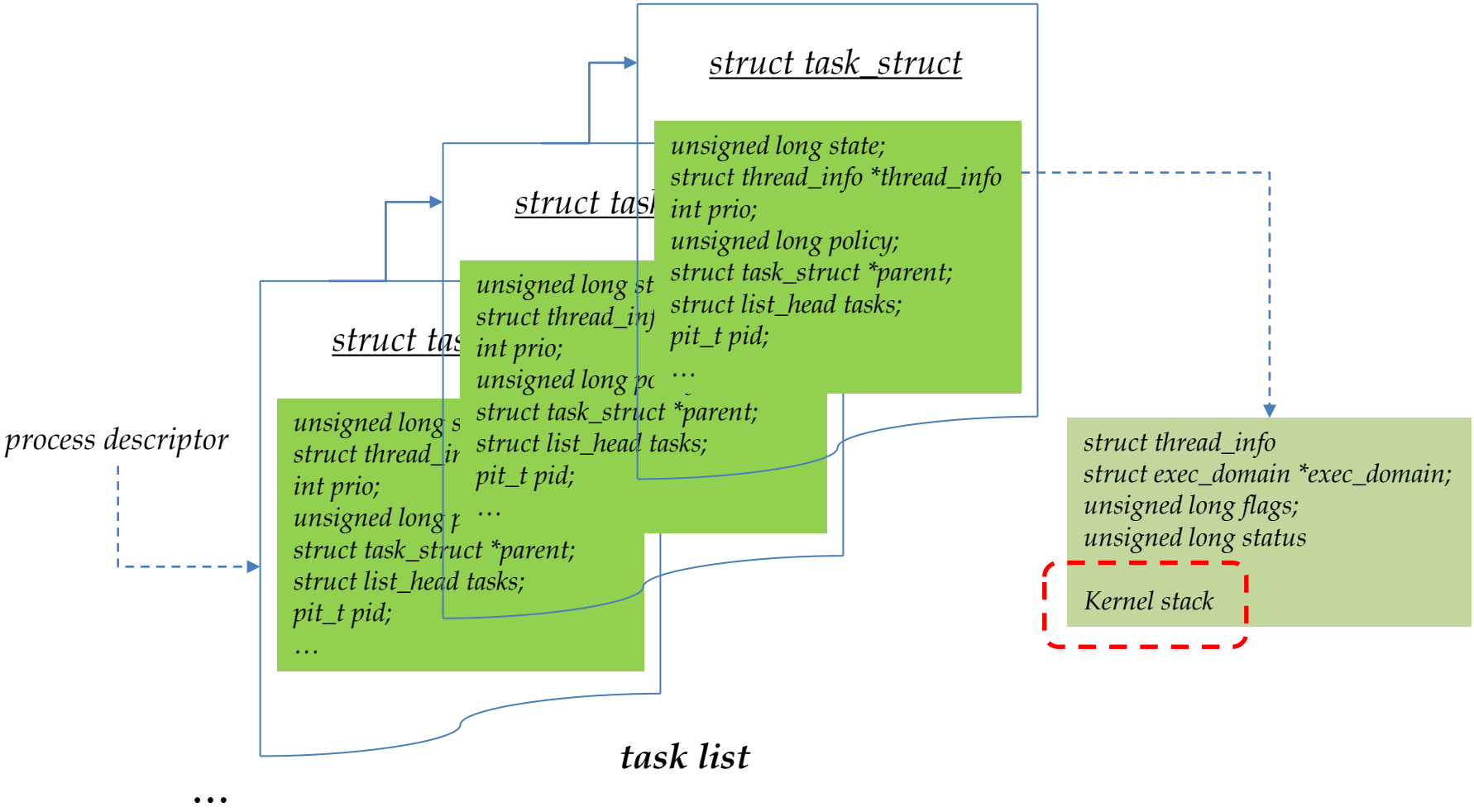
1. Task & Scheduling & Preemption(1) – *task scheduling**

(*) task, wait queue, run queue 간의 관계를 보여 주는 그림으로, wake_up 함수가 호출되면, 대기 중이던 해당 task가 run queue로 이동하여 CPU 를 할당 받게 된다(Scheduler가 그 역할을 담당함).
(*) wait queue, run queue로의 이동은 scheduler가 수행하기도 하지만, kernel code(my code) 내에서 명시적으로 수행하는 경우도 많다^^.



(*) CFS scheduler의 time complexity는 O(1)임
(CFS: Completely Fair Scheduler)
- Time accounting, process selection, sleeping & waking up

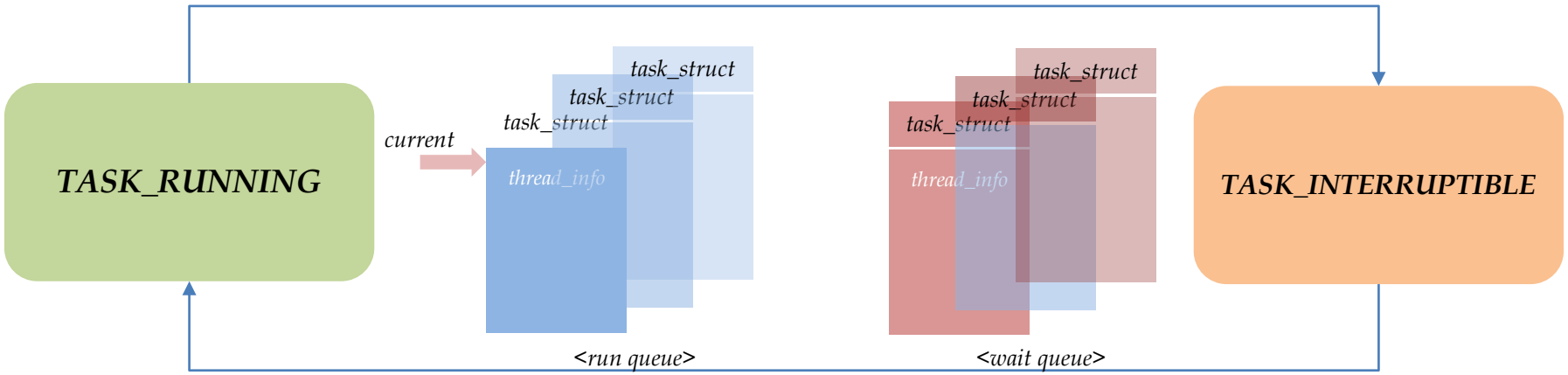
1. Task & Scheduling & Preemption(2) – *task**



(*) linux에서의 기본적인 실행 단위인, 프로세스를 위한 각종 정보를 담기 위해, task_struct가 사용되고 있음.
*(*) task_struct는 프로세스와 관련된 다양한 자원 정보를 저장하고, 커널 코드를 실행하기 위한 스택과 저수준의 flag는 thread_info structure에 저장됨^^*
(*) 참고로, task_struct data structure는 32bit CPU 기준으로 약 1.7KB의 크기(매우 큼)를 필요로 함.

1. Task & Scheduling & Preemption(3) – *task**

1) *add_wait_queue는 task를 wait queue에 추가하고, task의 상태를 TASK_INTERRUPTIBLE로 변경시킴.*
2) *이어 호출되는 schedule() 함수는 runqueue에 있는 가장 우선순위가 높은 다른 task를 CPU에게 할당해 줌.*



1) *Task가 기다리던 event가 발생하면, try_to_wake_up() 함수는 task의 상태를 TASK_RUNNING으로 변경시키고, activate_taks()함수를 호출하여 task를 runqueue에 추가시킴.*
2) *__remove_wait_queue는 task를 wait queue에서 제거함.*

(*) task 관련 queue로는 wait queue와 run queue가 있으며, run queue에 등록된 task는 실행되고, wait queue에 등록된 task는 특정 조건이 성립될 때까지 기다리게 된다.
(*) 위에서 언급된 특정 함수는 버전에 따라 차이가 있을 수 있음. 단, 전체적인 개념은 동일함.

1. Task & Scheduling & Preemption(4) - *sleeping & waking up**

(*) 아래 코드 style은 kernel code 이곳 저곳에서 매우 많이 사용되므로 눈여겨 볼 것^^
(*) 4장의 synchronization(11 – sleeping & wait queue) 내용과도 일맥 상통하는 부분임^^.

/* 아래 코드에서 'q'는 wait queue 임 */

schedule( ) : runqueue에서 가장 우선 순위가 높은 process(task)를 하나 꺼내어, 그것을 CPU에게 할당해 주는 것을 의미(scheduler가 작업해 줌). 여러 kernel routine에 의해서 직/간접적으로 호출됨.

DEFINE_WAIT(wait);
➔ 매크로를 이용하여 wait queue entry를 하나 생성함.

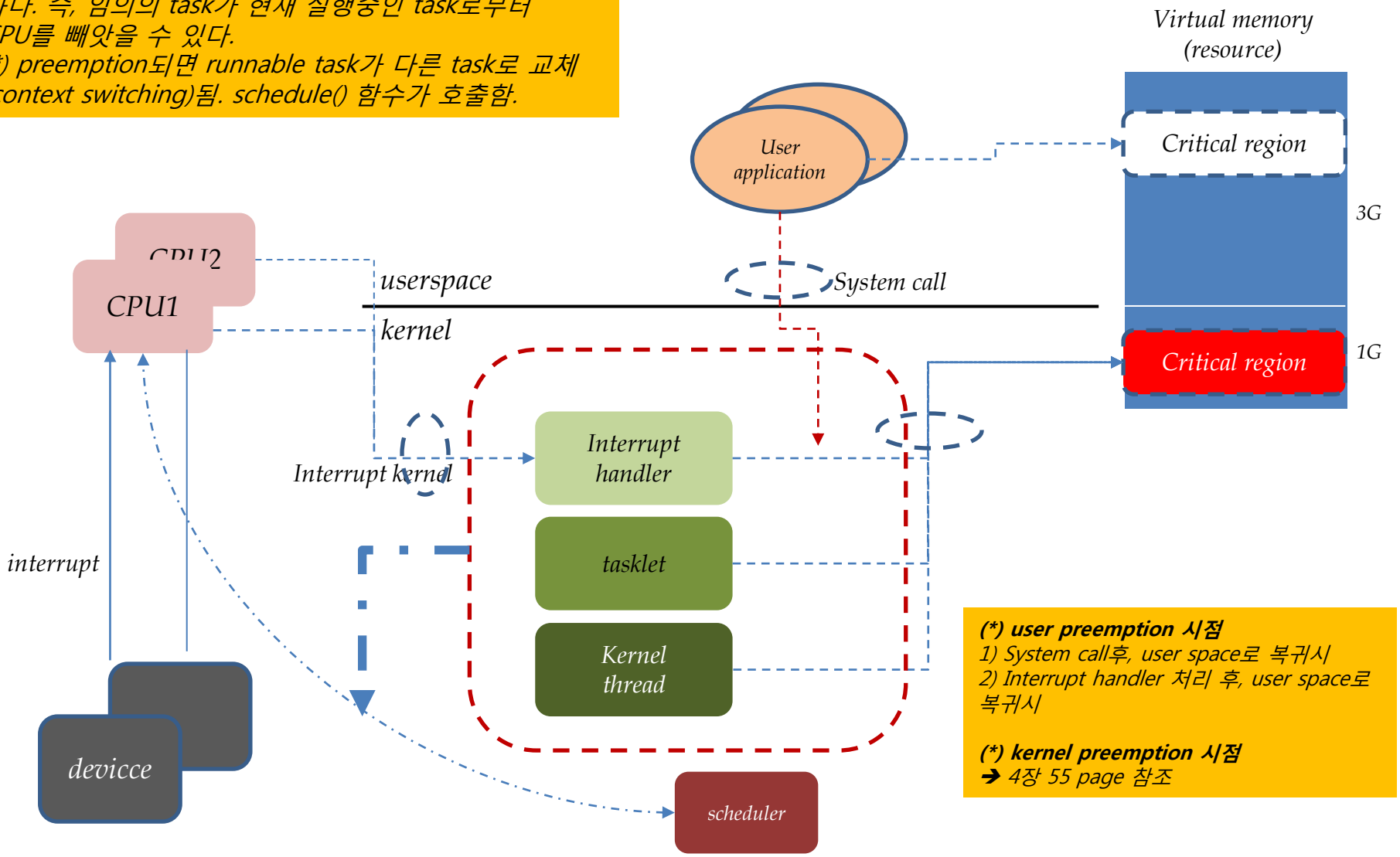
add_wait_queue(q, &wait);
➔ 자신을 wait queue에 넣는다. Wait queue는 깨어날 조건이 발생할 때, 해당 process를 깨운다.

```
while (!condition) {    /* condition is the event that we are waiting for */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
        ➔ process 상태를 TASK_INTERRUPTIBLE로 바꾼다.
    if (signal_pending(current)
            /* handle signal  & break the while loop*/
            ➔ wakeup signal이 도착하면, process는 깨어날 수 있는 상태가 된다.
              여기서 signal을 처리하고, while loop을 빠져 나오는 코드를 추가함(보통).
    schedule();
     ➔ wakeup할 조건이 성립되지 않으면, 해당 process는 여전히 sleep한다.
       (즉, 다른 process가 CPU를 사용할 수 있도록 schedule 함수를 호출)
}
```

finish_wait(&q, &wait);
➔ 깨어 나기 위해 자신을 wait queue에서 제거한다(runqueue로 이동함. Scheduler에 의해 CPU를 할당 받을 준비 !).

1. Task & Scheduling & Preemption(5) – ***preemption****

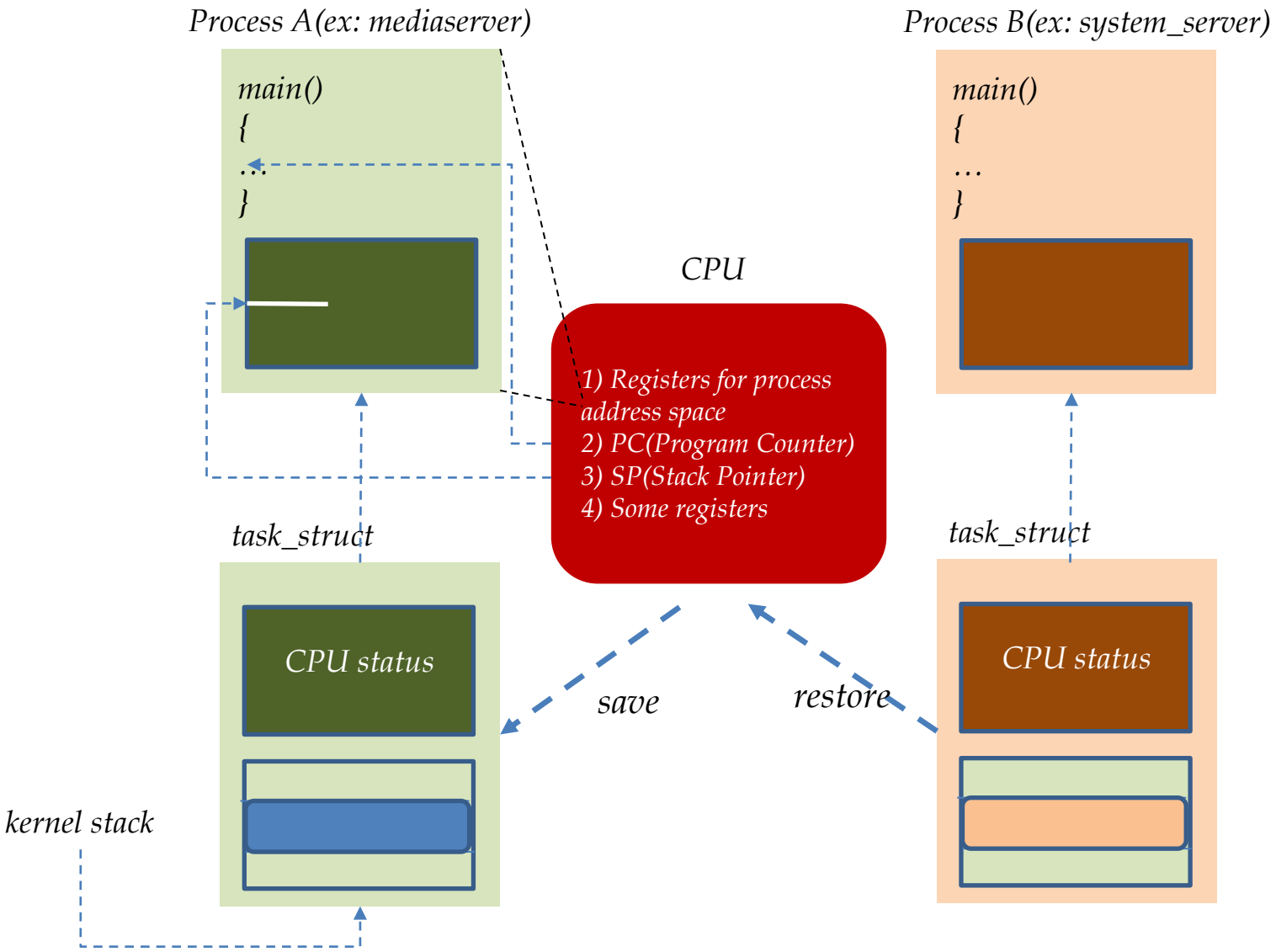
(*) linux kernel(2.6 이상)은 interrupt, system call, multiprocessor등의 다양한 상황에서 ***fully preemptive*** *하다. 즉, 임의의 task가 현재 실행중인 task로부터 CPU를 빼앗을 수 있다.*
(*) preemption되면 runnable task가 다른 task로 교체(context switching)됨. schedule() 함수가 호출함.



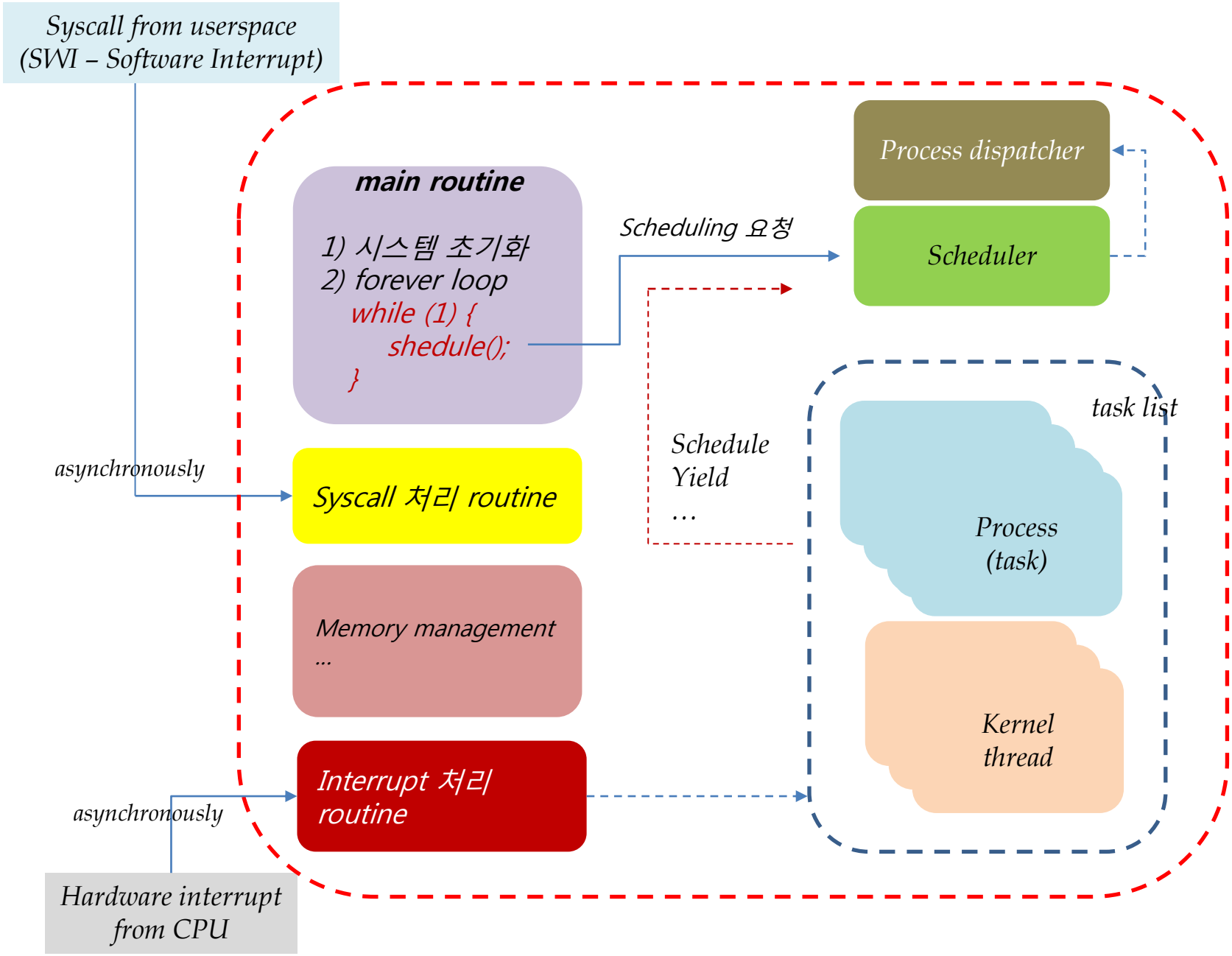
(*) user preemption 시점
1) System call후, user space로 복귀시
2) Interrupt handler 처리 후, user space로 복귀시

(*) kernel preemption 시점
➔ 4장 55 page 참조

1. Task & Scheduling & Preemption(6) – ***context switching***



1. Task & Scheduling & Preemption(7) – ***linux kernel***



다음 장으로 넘어가기 전에 …*

(*) 앞으로 설명할 task/scheduling, top/bottom halves, timer routine 등은 모두 아래와 같은 형태로 일반화시켜 생각해 볼 수 있을 듯하다.
*➔ 너무 일반화 시켰나 ^^*
(*) 한가지 재밌는 것은 이러한 구조는 kernel 내에서 뿐만 아니라, Android UI 내부 Message 전달 구조 및 media framework의 핵심인 stagefright event 전달 구조에서도 유사점을 찾을 수 있다는 것이다^^.

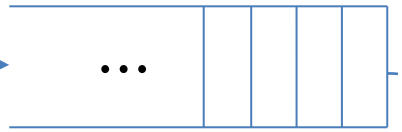
<work 정의>

1) task
2) some function routines
: interrupt handler,
softirq, tasklet,
work queue, timer function

<실행 요청>

1) schedule
2) Interrupt
3) it's own schedule func

주기 혹은 비주기적인 반복 요청
(이게 없으면, 한번만 실행되고 맘)



w/, w/o queue

1) runqueue, waitqueue
2) work queue
3) irq queue(interrupt)
4) tasklet queue
5) Timer queue…

<queue에서 가져옴>

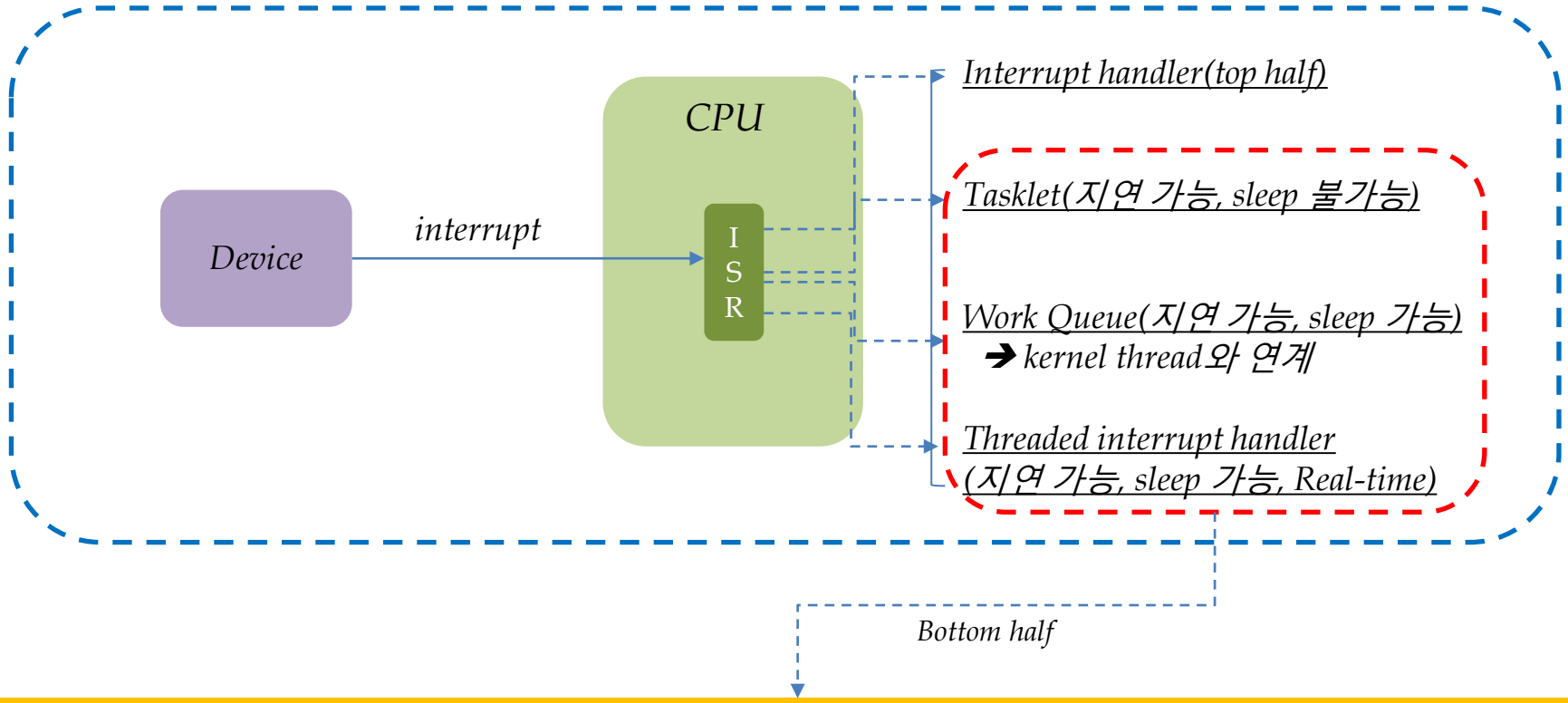
<처리 루틴>

1) scheduler
2) interrupt handling,
tasklet processing,
timer processing,
3) worker thread,
Thread for threaded
Interrupt handler

2. Top Half, Bottom Halves and Deferring Work – ***개념****

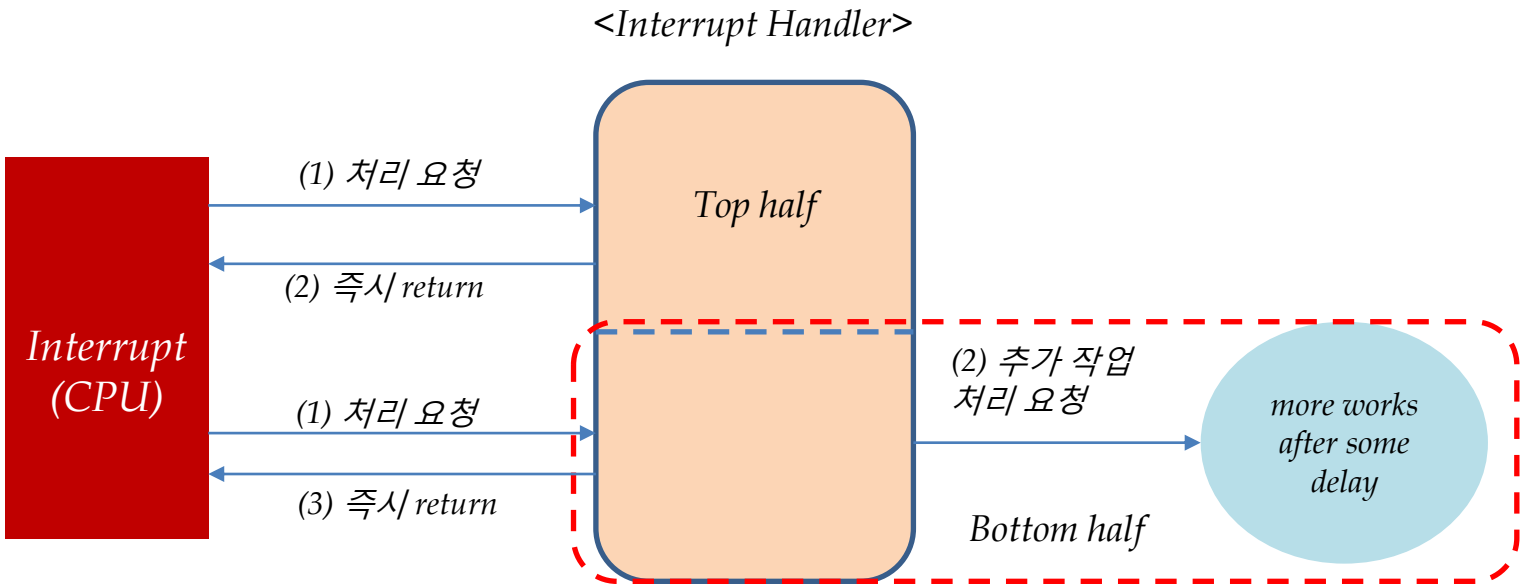
- 1) Interrupt handler를 top half라고 하며, 지연 처리(deferring work)가 가능한 루틴을 bottom half라고 함.
  - ➔ *지연 처리는 interrupt context(tasklet)에 대해서도 필요하며, process context(work queue)에도 필요하다.*
- 2) (bottom half 중에서도)해당 작업이 sleep 가능하거나 sleep이 필요할 경우: **work queue 사용**
  - ➔ *process context에서 실행*
- 3) 1의 조건에 해당하지 않으며 빠른 처리가 필수적인 경우: **tasklet 사용**
  - ➔ *interrupt context에서 실행*
  - ➔ *Softirq도 tasklet과 동일한 구조이나, 사용되는 내용이 정적으로 정해져 있음. 따라서 programming 시에는 동적인 등록이 가능한 tasklet이 사용된다고 이해하면 될 듯^^*
- 4) tasklet과 softirq의 관계와 마찬가지로, work queue는 kernel thread를 기반으로 하여 구현되어 있음.
- 5) Threaded interrupt handler를 사용하면, real-time의 개념이 들어간 thread 기반의 interrupt handling도 가능하다.
  - ➔ *work queue와는 달리, 우선 순위가 높은 interrupt 요청 시, 빠른 처리가 가능하다.*
  - ➔ *work queue가 있음에도 이 개념이 등장한 이유는, interrupt handler 내에서 처리할 작업은 시간이 오래 소요되지만, 마치 top half처럼 바로 처리할 수 없을까 하는 생각(요구)에서 나온 듯 함^^.*

2. Top Half, Bottom Halves and Deferring Work – ***개념****



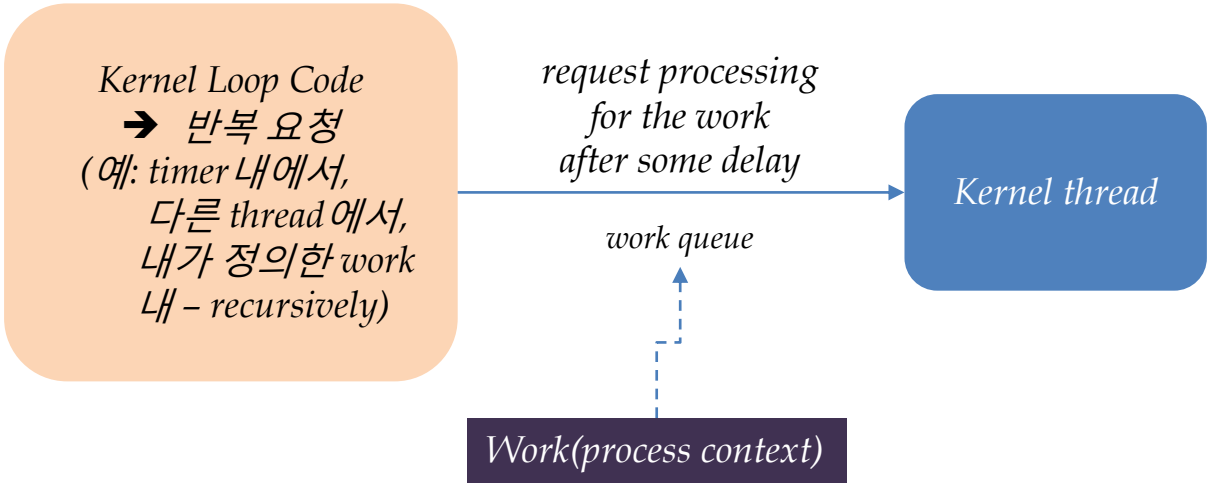
(*) 위의 그림에서 처럼, tasklet, work queue, threaded interrupt handler routine 모두 interrupt handler 내에서 지연 처리를 위해 사용될 수 있는 방식들이다.
(*) 다만, tasklet은 interrupt context에서 수행되며, work queue 및 threaded interrupt handler는 process context에서 수행되므로, 지연시킬 작업의 내용을 보고, 어떤 방식을 사용해야 할 지 결정해야 한다.
(*) work queue는 bottom half 개념으로 등장하기는 했으나, 위에서와 같이 interrupt handler 내에서의 지연 처리 뿐만 아니라, 임의의 process context에 대한 지연 처리 시에도 널리 활용되고 있다.

2. Top Half, Bottom Halves and Deferring Work – ***interrupt context 지연****



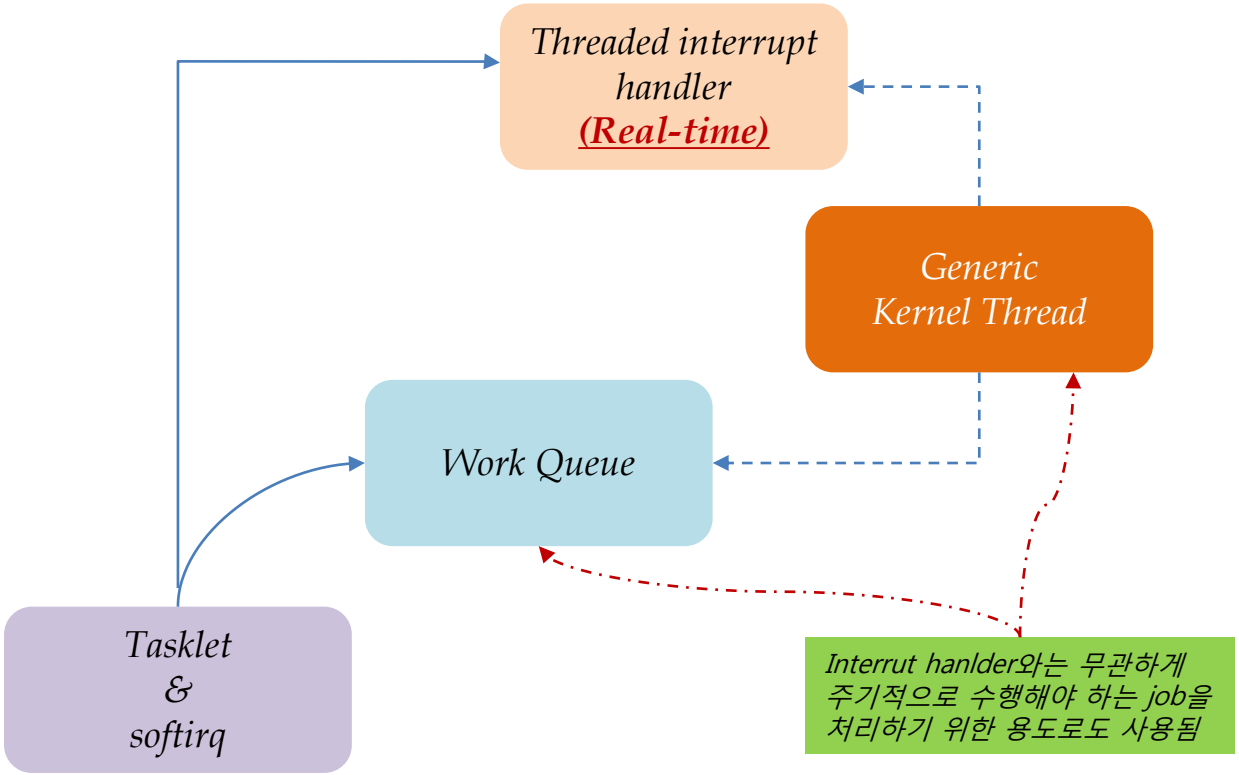
(*) Top half의 경우는 바로 처리 가능한 interrupt handler를 의미하며, Bottom half는 시스템의 반응성 (interrupt 유실 방지)을 좋게 하기 위하여, 시간이 오래 걸리는 작업을 별도의 루틴을 통해 나중에 처리하는 것을 일컫는다.
*➔어쨌거나, interrupt handler내에서 처리할 작업이 좀 있는 경우에는 bottom half 처리 루틴에게 일을 넘겨 주고, 자신은 빨리 return하므로써, 다음 interrupt의 유실을 최대한 막을 수 있는 것으로 이해하면 될 듯 ^^*

2. Top Half, Bottom Halves and Deferring Work – *process context 지연*



*(*) softirq/tasklet이 interrupt 처리 지연과 관련이 있다면, work queue는 process context 지연과 관련이 있다. 복수개의 요청(process context)을 work queue에 등록해 둔 후, 나중(after some delay)에 처리하는 것으로 효율을 향상시킬 목적으로 사용됨^^*
(*) 따라서 앞서 이미 언급한 바와 같이, work queue의 경우는 interrupt handler 내에서의 지연 처리 뿐만 아니라, 임의의 process context에 대한 지연 처리에도 널리 활용되고 있다.

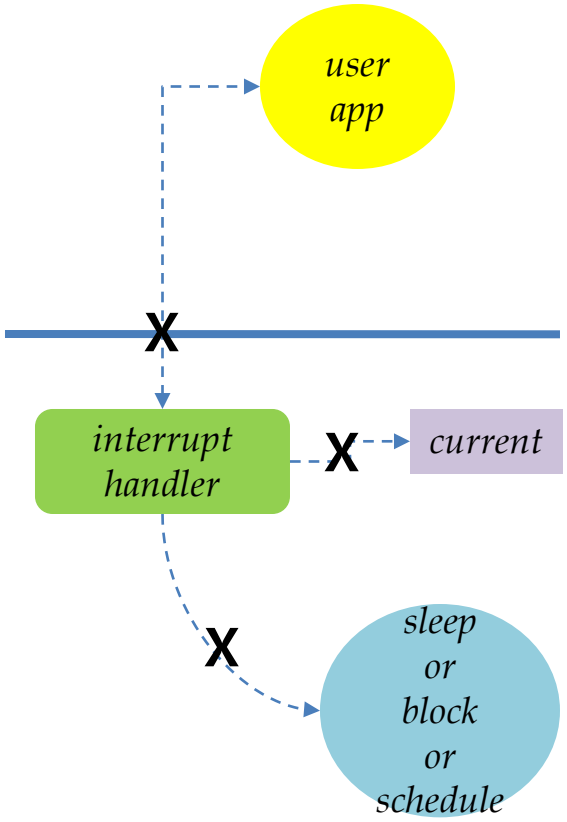
2. Top Half, Bottom Halves and Deferring Work – ***bottom halves****



(*) 위의 화살표가 특별한 의미를 부여하는 것은 아님. 다만, deferring work 관련하여 대략적으로 위와 같이 발전(진전)하고 있는 것으로 보이며, 따라서 본 문서에서도 위의 순서를 따라 설명을 진행하고자 함.
(*) 가장 최근에 등장한 Threaded interrupt handler의 경우는 real-time OS의 특징(실시간 처리)을 지향하고 있다.

2. Top Half, Bottom Halves and Deferring Work – ***interrupt & process context/1****

- **<interrupt context(=atomic context)의 제약 사항>**



(*) user space로의 접근이 불가능하다. Process context가 없으므로, 특정 process와 결합된 user space로 접근할 방법이 없다(예: copy_to_user(), copy_from_user() 등 사용 불가)

(*) current 포인터(현재 running 중인 task pointer)는 atomic mode에서는 의미가 없으며, 관련 코드가 interrupt 걸린 process와 연결되지 않았으므로, 사용될 수 없다(current pointer에 대한 사용 불가).

(*) sleeping이 불가하며, scheduling도 할 수 없다. Atomic code는 schedule() 함수를 call해서는 안되며, wait_event나 sleep으로 갈 수 있는 어떠한 형태의 함수를 호출해서도 안된다. 예를 들어, kmalloc(..., GFP_KERNEL) 을 호출해서는 안된다(GFP_ATOMIC을 사용해야 함). Semaphone도 사용할 수 없다(down 사용 불가. 단, up이나 wake_up 등 다른 쪽을 풀어주는 코드는 사용 가능함)

(*) 위의 내용은 앞으로 설명할 interrupt handler와 tasklet에 모두 적용되는 내용임.
(*) 반대로, work queue는 process context에서 동작하므로 위에서 제약한 사항을 모두 사용할 수 있음^^.

2. Top Half, Bottom Halves and Deferring Work – ***interrupt & process context/2****

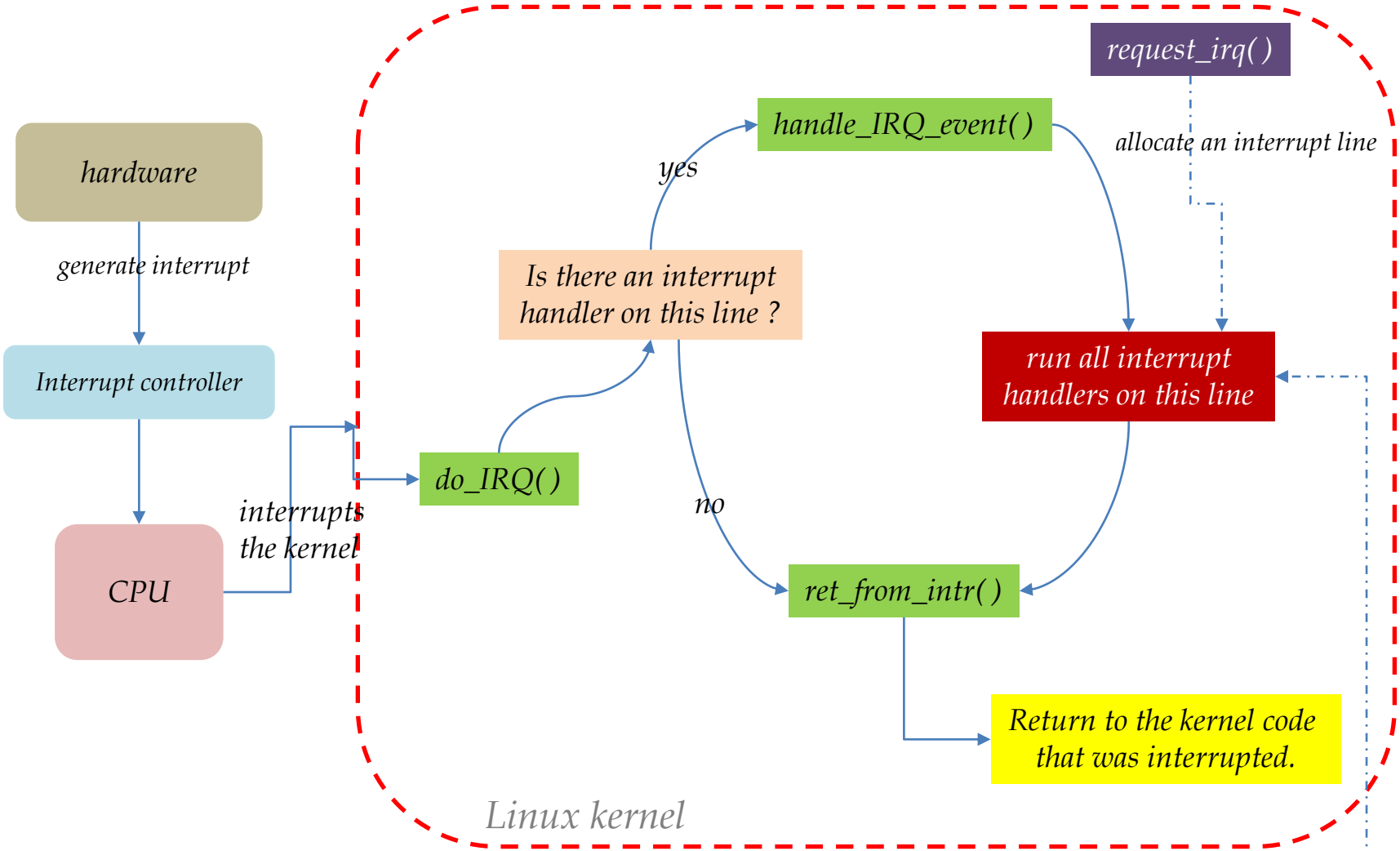
(*) user application에 의해 발생하는 system call을 처리하는 kernel code의 경우 process context에서 실행된다고 말함.
➔ process context에서 실행되는 kernel code는 다른 kernel code에 의해 CPU 사용을 빼앗길 수 있다(preemptive)
*➔ **process context 대상**: user process로 부터 온 system call 처리, work queue, kernel thread, threaded interrupt handler*

(*) 반면에 interrupt handler(얘가 전부는 아님)를 interrupt context라고 이해하면 쉬울 듯.
➔ interrupt context에서 수행되는 kernel code는 끝날 때까지 다른 kernel code에 의해 중단될 수 없다.
*➔ **interrupt context 대상**: hard interrupt handler, softirq/tasklet*

Interrupt context에서 해서는 안되는 일

1) Sleep하거나, processor를 포기
2) Mutex 사용
3) 시간을 많이 잡아 먹는 일
4) User space(virtual memory) 접근

2. Top Half, Bottom Halves and Deferring Work - ***Interrupt Handler(top half)****



(*) 보통은 1개의 handler를 처리하겠으나, shared IRQ인 경우는 동시에 여러 handler가 몰릴 수 있다.

2. Top Half, Bottom Halves and Deferring Work - ***Interrupt Handler(top half)****

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev);
```

➔ *Interrupt handler 등록 및 실행 요청*

➔ *irq(첫번째 argument)가 interrupt number 임.*

<두 번째 argument handler>

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

(*) /proc/interrupts에서 인터럽트 상태를 확인할 수 있음 !

Interrupt handler

H/W interrupt가 발생할 때마다 호출됨

*) 인터럽트 처리 중에 또 다른 인터럽트가 들어 올 수 있으니, 최대한 빠른 처리가 가능한 코드로 구성하게 됨.

synchronize_irq()
➔*free_irq를 호출하기 전에 호출하는 함수로, 현재 처리 중인 interrupt handler가 동작을 완료하기를 기다려 줌.*

free_irq()
➔ *인터럽트 handler 등록 해제 함수*

disable_irq()
➔ *해당 IRQ 라인에 대한 interrupt 리포팅을 못하도록 함.*

disable_irq_nosync()
➔ *Interrupt handler가 처리를 끝내도록 기다리지 않고 바로 return 함*

enable_irq()
➔ *해당 IRQ 라인에 대한 interrupt 리포팅을 하도록 함.*

2. Top Half, Bottom Halves and Deferring Work - ***Interrupt Handler(top half)***

<Shared handler 구현 시 요구 사항>

**) 한 개의 interrupt line을 여러 장치가 공유(따라서, interrupt handler도 각각 서로 다름)할 경우에는 좀 더 특별한 처리가 요구된다.*

1) request_irq( )함수의 flags 인자로 IRQF_SHARED를 넘겨야 한다.

2) request_irq( )함수의 dev 인자로는 해당 device 정보를 알려 줄 수 있는 내용이 전달되어야 한다. NULL을 넘겨주면 안된다.

3) 마지막으로 interrupt handler는 자신의 device가 실제로 interrupt를 발생시켰는지를 판단할 수 있어야 한다. 이를 위해서는 handler 자체만으로는 불가능하므로, device에서도 위를 위한 방법을 제공해야 하며, handler도 이를 확인하는 루틴을 제공해야 한다.

Kernel이 interrupt를 받으면, 등록된 모든 interrupt handler를 순차적으로 실행하게 된다. 따라서, interrupt handler 입장에서는 자신의 device로 부터 interrupt가 발생했는지를 판단하여, 그렇지 않을 경우에는 재빨리 handler 루틴을 끝내야 한다.

2. Top Half, Bottom Halves and Deferring Work - ***Interrupt Handler(top half)****

<Interrupt – Disable/Enable>

**) 드라이버로 하여금, interrupt line으로 들어오는 interrupt를 금지 및 다시 허용하는 것이 가능한데, interrupt를 disable하게 되면, 처리 중인 resource를 보호할 수 있다.*

**) interrupt handler를 수행하기 직전에 kernel이 알아서 interrupt를 disable해 주고, handler를 수행한 후에 interrupt를 다시 enable시켜 주므로, handler routine내에서는 interrupt를 disable해 줄 필요가 없다.*

```
disable_irq(irq);          ← system의 모든 processor로 부터의 interrupt를 금지시킴
                             (해당 IRQ line에 대해서만)
local_irq_save(flags);     ← 현재 상태 저장

    handler(irq, dev_id);  ← interrupt handler 루틴 구동

local_irq_restore(flags);  ← 저장된 상태 복구
enable_irq(irq);           ← system의 모든 processor로 부터의 interrupt를  허용함
```

(*) enable_irq/disable_irq는 항상 쌍으로 호출되어야 한다. 즉, disable_irq를 두번 호출했으면 Enable_irq도 두번 호출해 주어야 금지된 interrup가 해제됨.

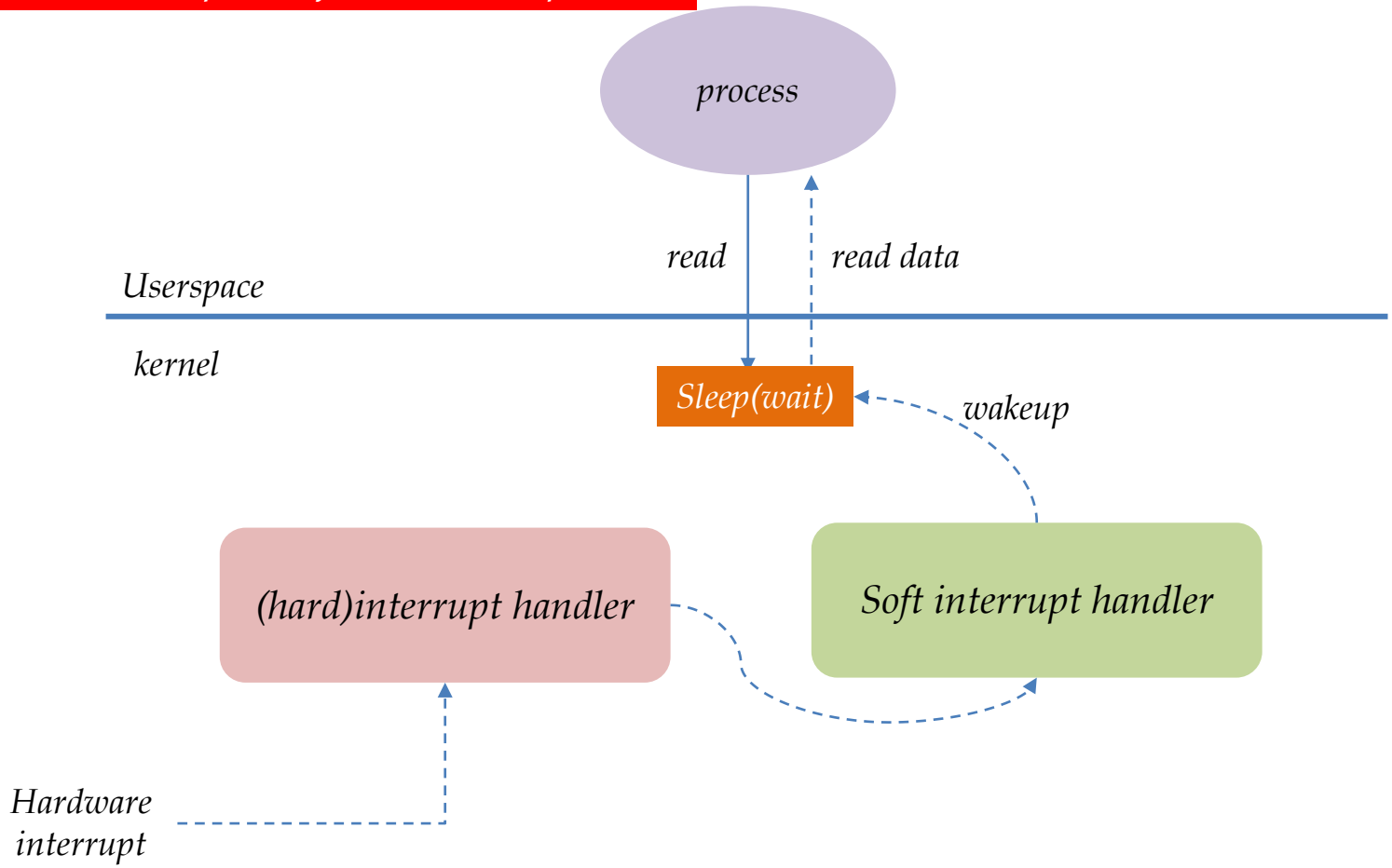
```
local_irq_disable();       ← 현재 processor 내부에서만 interrupt를 금지시켜줌.
/* interrupts are disabled */
local_irq_enable();
```

2. Top Half, Bottom Halves and Deferring Work - *Interrupt Handler(top half)*

Interrupt 관련 함수	함수의 의미
local_irq_disable()	Local(같은 processor 내) interrupt 금지.
local_irq_enable()	Local interrupt 허용
local_irq_save()	Local interrupt의 현재 상태 저장 후, interrupt 금지
local_irq_restore()	Local interrupt의 상태를 이전 상태로 복구
disable_irq()	주어진 interrupt line(전체 processor에 해당)에 대한 interrupt 금지. 해당 line에 대해 interrupt가 발생하지 않는 것으로 보고 return함.
disable_irq_nosync()	주어진 interrupt line(전체 processor에 해당)에 대한 interrupt 금지
enable_irq()	주어진 interrupt line(전체 processor에 해당)에 대한 interrupt 허용
irqs_disabled()	Local interrupt가 금지되어 있으면 0이 아닌 값 return, 그렇지 않으면 0 return.
in_interrupt()	현재 코드가 interrupt context내에 있으면, 0이 아닌 값 return, process context에 있으면 0 return.,
in_irq()	현재 interrupt handler를 실행 중이면, 0이 아닌 값 return, 그렇지 않으면 0 return.

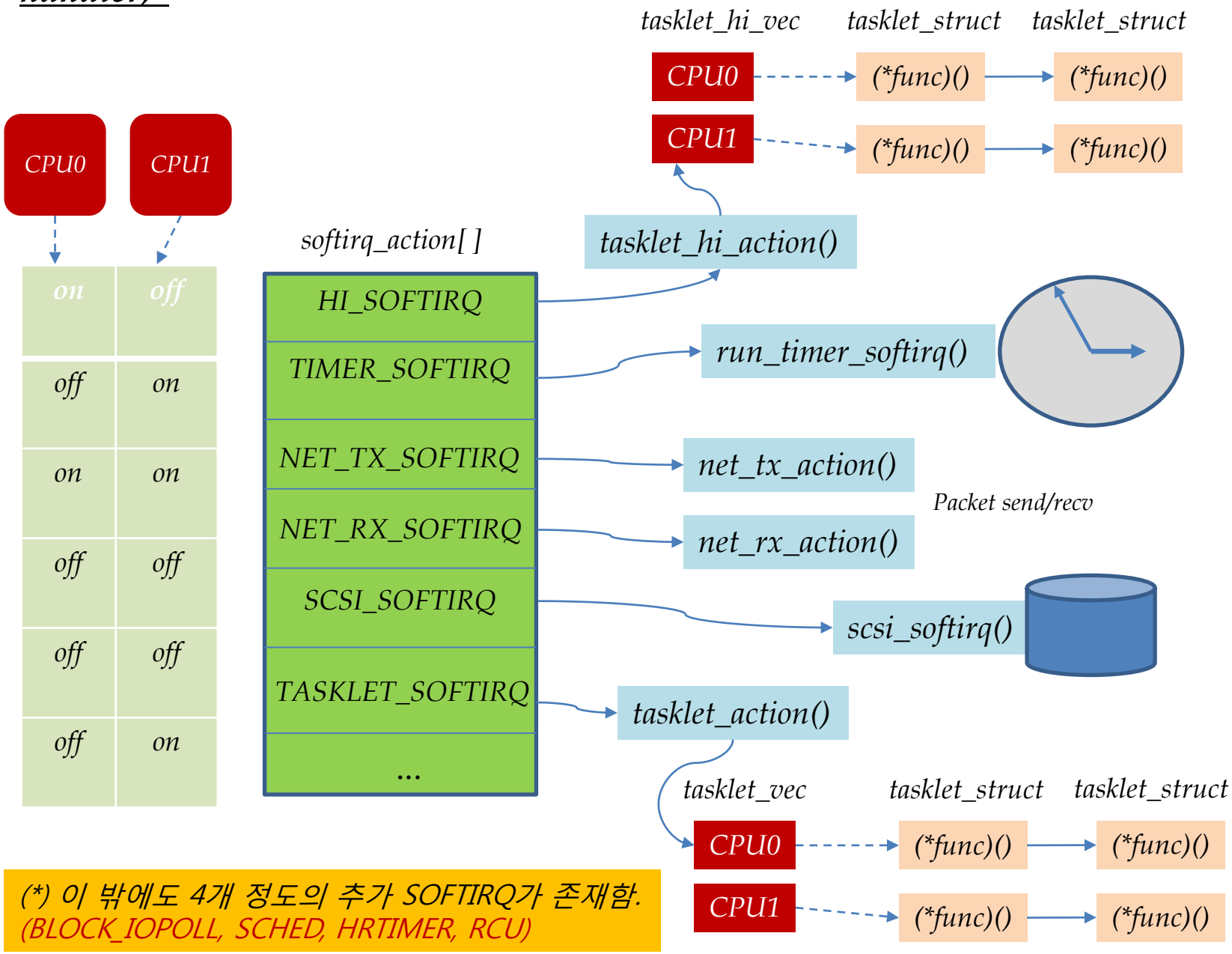
2. Top Half, Bottom Halves and Deferring Work – ***Softirqs(soft interrupt handler)****

hardware interrupt와 software interrupt의 예



(*) linux에는 두 종류의 software interrupt가 있다.
➔ bottom half 개념의 soft interrupt와 system call(SWI) !!!
➔ 본 장에서 언급하는 내용은 bottom half 개념의 soft interrupt 임.

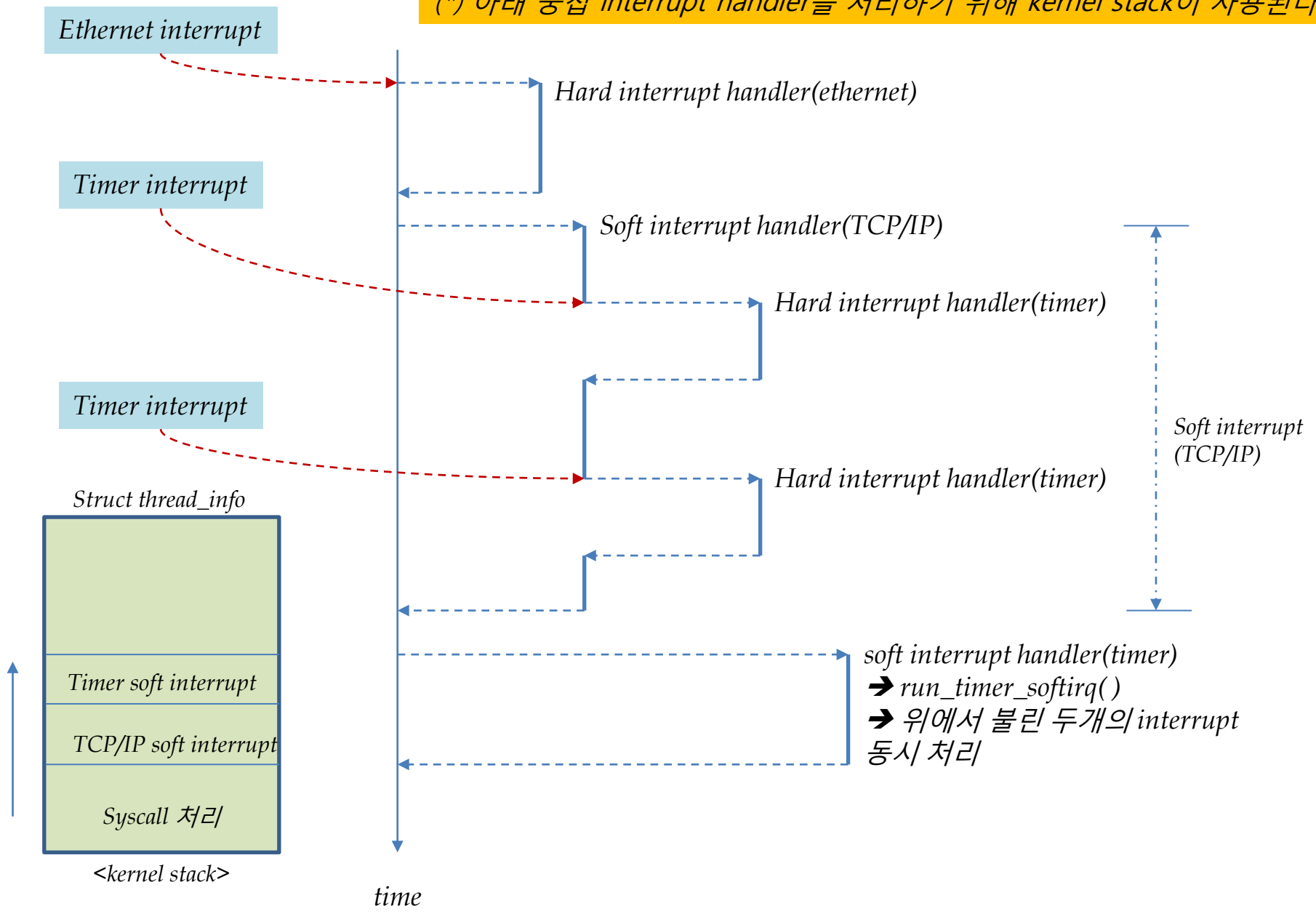
2. Top Half, Bottom Halves and Deferring Work – ***Softirqs(soft interrupt handler)****



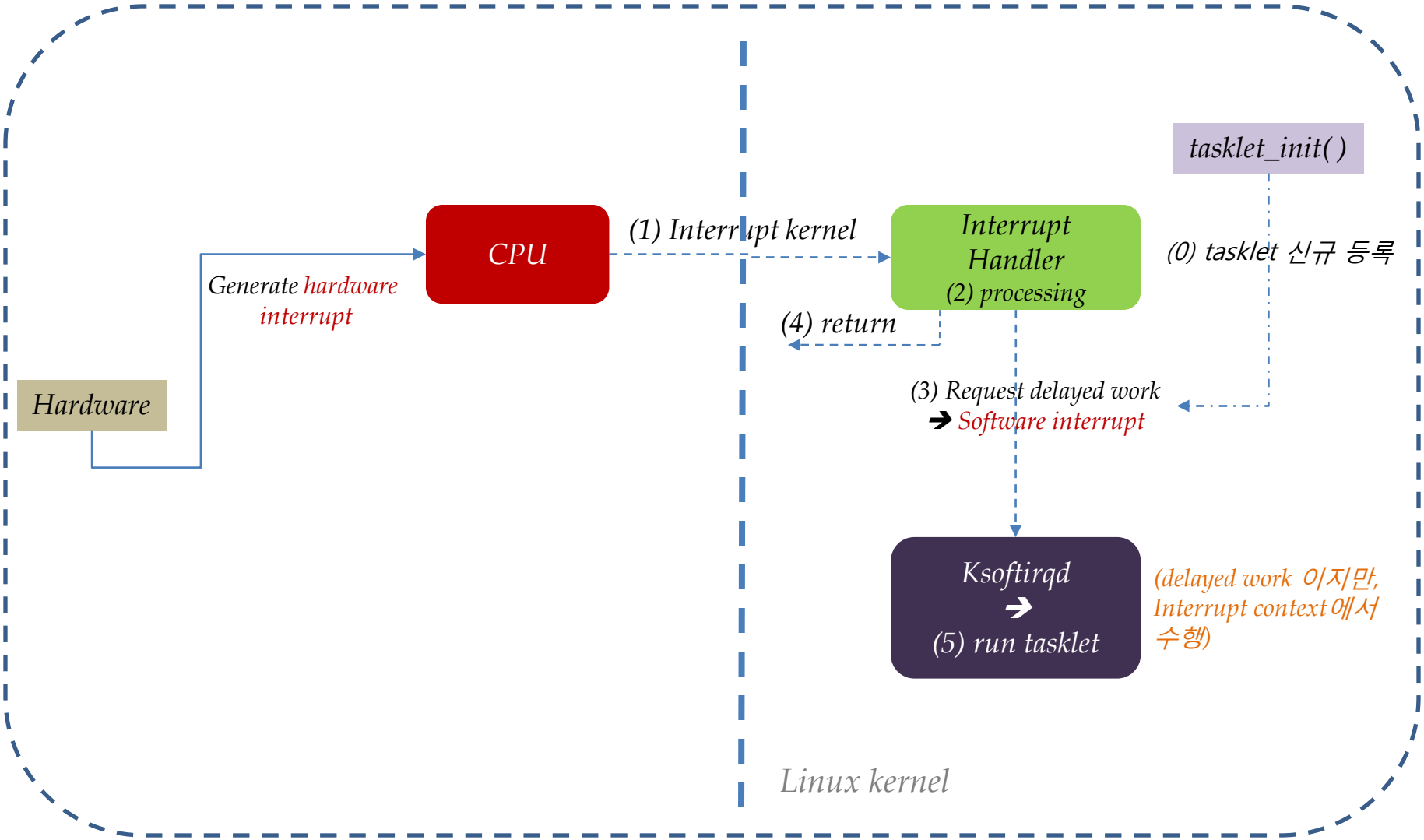
(*) 이 밖에도 4개 정도의 추가 SOFTIRQ가 존재함.
(BLOCK_IOPOLL, SCHED, HRTIMER, RCU)

2. Top Half, Bottom Halves and Deferring Work – ***Softirqs(soft interrupt handler)****

(*) 아래 중첩 interrupt handler를 처리하기 위해 kernel stack이 사용된다.

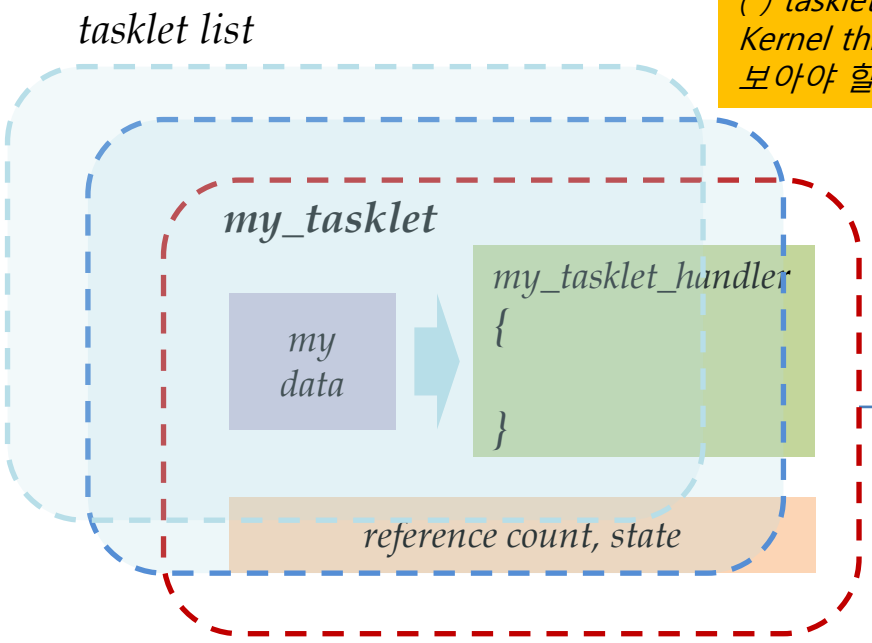


2. Top Half, Bottom Halves and Deferring Work – ***Tasklet****



2. Top Half, Bottom Halves and Deferring Work – ***Tasklet****

(*) tasklet과 softirq의 동작 원리는 동일함. 다만, softirq는 compile-time에 이미 내용(action)이 정해져 있으며, tasklet은 dynamic하게 등록할 수 있는 형태임.
(*) tasklet은 동시에 하나씩만 실행됨(count와 state 값을 활용)
➔ 이는 multi-processor 환경에서도 동일하게 적용됨.
(*) tasklet은 task 개념과는 전혀 무관하며, 또한 work queue와는 달리 Kernel thread를 필요로 하지 않음(그 만큼 간단한 작업을 처리한다고 보아야 할 듯^^).



tasklet_schedule(&my_tasklet)
➔ 이것이 호출되면 tasklet handler 실행됨

tasklet_init(&my_tasklet, my_tasklet_handler)
or
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, my_data)
➔ 초기화

(*) tasklet_enable(&my_tasklet)
➔ disable된 tasklet를 enable 시킬 때 사용
(*) tasklet_disalbe(&my_tasklet)
➔ enable된 tasklet를 disable 시킬 때 사용
(*) tasklet_kill( )
➔ tasklet를 pending queue에서 제거할 때 사용

my_tasklet_handler(my_data) will be run !
➔ 얘는 빠르게 처리되는 코드이어야 함 !

2. Top Half, Bottom Halves and Deferring Work – ***Tasklet****

(*) 아래 code는 softirq 및 tasklet을 실제로 처리해 주는 ksoftirqd kernel thread의 메인 루틴을 정리한 것임.
(*) softirq or tasklet이 발생할 때마다 실행하게 되면, kernel 수행이 바빠지므로, user space process가 처리되지 못하는 문제(starvation)가 있을 수 있으며, interrupt return 시마다 실행하게 되면, softirq(tasklet) 처리에 문제(starvation)가 발생할 수 있어, 해결책으로써, ksoftirqd kernel thread를 두어 처리하게 됨.
(*) ksoftirqd는 평상시에는 낮은 우선순위로 동작하므로, softirq/tasklet 요청이 많을 지라도, userspace가 starvation 상태로 빠지는 것을 방지하며, system이 idle 상태인 경우에는 kernel thread가 즉시 schedule되므로, softirq/tasklet을 빠르게 처리할 수 있게 된다.

ksoftirqd의 메인 루틴

```
for (;;) {
    if (!softirq_pending(cpu))
        schedule( );                    /* softirq/tasklet 요청이 없으면, sleep */
    set_current_state(TASK_RUNNING);
    while (softirq_pending(cpu)) {      /* softirq stack의 내용 실행 */
        do_softirq( );
        if (need_resched( ))
            schedule( );
    }
    set_current_state(TASK_INTERRUPTIBLE);
}
```

2. Top Half, Bottom Halves and Deferring Work - ***Tasklet***

<Tasklet scheduling 절차>

1. *Tasklet의 상태가 TASKLET_STATE_SCHED 인지 확인한다. 만일 그렇다면, tasklet이 이미 구동하도록 schedule되어 있으므로, 아래 단계로 내려갈 필요 없이 즉시 return 한다.*

2. *그렇지 않다면, __tasklet_schedule( ) 함수를 호출한다.*

3. *Interrupt system의 상태를 저장하고, local interrupt를 disable시킨다. 이렇게 함으로써, tasklet_schedule( ) 함수가 tasklet를 조작할 때, 다른 것들과 엉키지 않게 된다.*

4. *Tasklet을 tasklet_vec(regular tasklet 용) 이나 tasklet_hi_vec(high-priority tasklet 용) linked list에 추가한다.*

5. *TASKLET_SOFTIRQ 혹은 HI_SOFTIRQ softirq를 발생(raise)시키면, 잠시 후 do_softirq( ) 함수에서 이 tasklet을 실행하게 된다.*
   - ➔ *do_softirq( )는 마지막 interrupt가 return할 때 실행하게 된다.*
   - ➔ *do_softirq( ) 함수 내에서는 tasklet processing의 핵심이라 할 수 있는 tasklet_action( ) 및 tasklet_hi_action() handler를 실행하게 된다.*
   - ➔ *이 과정을 다음 페이지에 상세하게 정리*

6. *Interrupt를 이전 상태로 복구하고, return 한다.*

2. Top Half, Bottom Halves and Deferring Work - ***Tasklet***

<Tasklet handler 수행 절차>

1. Local interrupt delivery를 disable 시킨 후, 해당 processor에 대한 tasklet_vec 혹은 tasklet_hi_vec 리스트 정보를 구해온다. 이후, list를 clear(NULL로 셋팅) 시킨 후, 다시 local interupt delivery를 enable 시킨다.

2. 1에서 얻은 list를 구성하는 각각의 (pending) tasklet에 대해 아래의 내용을 반복한다.

3. CPU가 두개 이상인 system이라면, tasklet이 다른 processor 상에서 동작 중인지 체크한다 (TASKLET_STATE_RUN 플래그 사용). 만일 그렇다면, tasklet을 실행하지 않고, 다음번 tasklet을 검사한다.

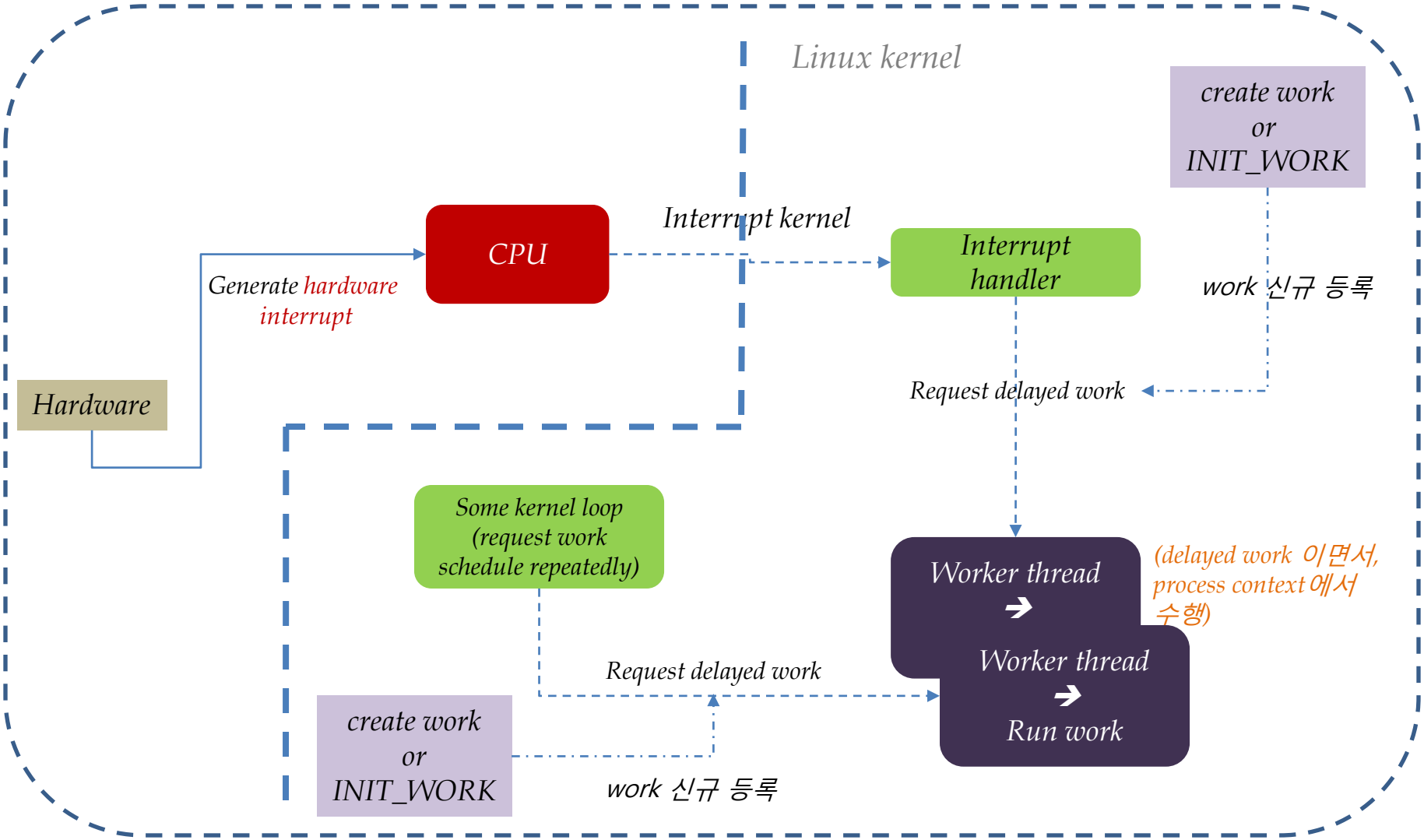
4. Tasklet이 실행되지 않고 있으면, TASKLET_STATE_RUN 플래그 값을 설정한다. 그래야 다른 procssor가 이 tasklet을 실행하지 못하게 된다.

5. Tasklet이 disable되어 있지 않은지를 확인하기 위해 zero count 값을 검사한다. 만일 tasklet이 disable 되어 있으면, 다음 tasklet으로 넘어간다.

6. 이제 tasklet을 실행할 모든 준비가 되었으므로, tasklet handler를 실행한다.

7. Tasklet을 실행한 후에는 TASKLET_STATE_RUN 플래그를 clear한다.
8. 이상의 과정을 모든 pending tasklet에 대해 반복한다.

2. Top Half, Bottom Halves and Deferring Work - ***Work Queue****

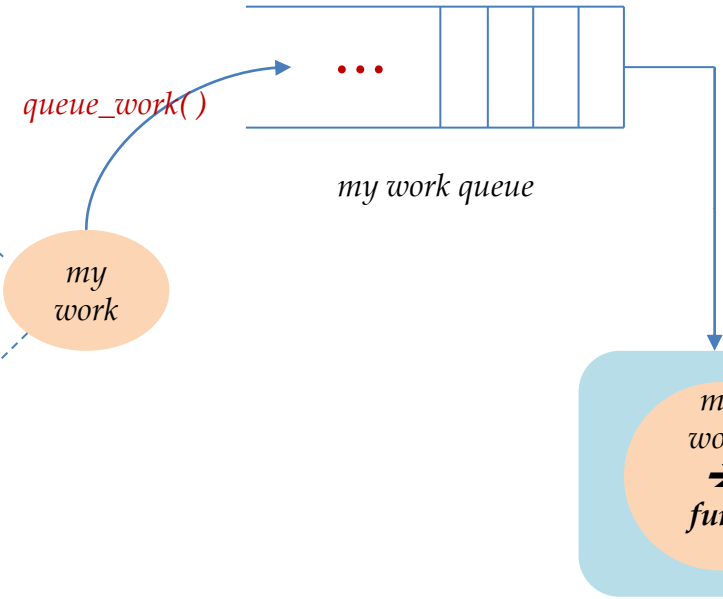


2. Top Half, Bottom Halves and Deferring Work - ***Work Queue(data structure)****

```
struct workqueue_struct {
   struct cpu_workqueue_struct cpu_wq[NR_CPUS];
   struct list_head list;
   const char *name;
   int singlethread;
   int freezeable;
   int rt;
};
```

```
struct cpu_workqueue_struct {
   spinlock_t lock;
   struct list_head worklist;
   wait_queue_head_t more_work;
   struct work_struct *current_struct;
   struct workqueue_struct *wq;
   task_t *thread;
};
```

<work queue 관련 data structure>




```
struct work_struct {
   atomic_long_t data;
   struct list_head entry;
   work_func_t func;
};
```

<work 관련 data structure>

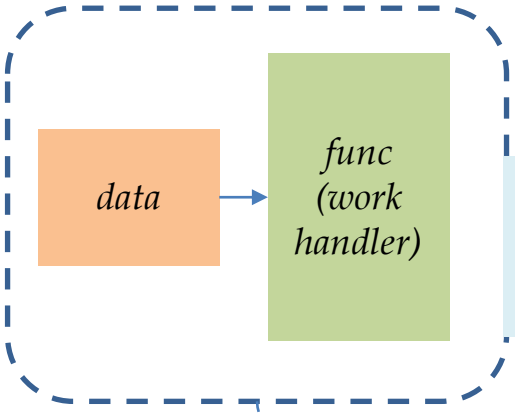
<worker thread flow>

1) Thread 자신을 sleep 상태로 만들고, wait queue에 자신을 추가한다.
2) 처리할 work이 없으면, schedule()을 호출하고, 자신은 여전히 sleep한다.
3) 처리할 work이 있으면, wakeup 상태로 바꾸고, wait queue에서 빠져나온다.
4) run_workqueue( ) 함수를 호출하여, deferred work을 수행한다.
➔ func( ) 함수 호출함.

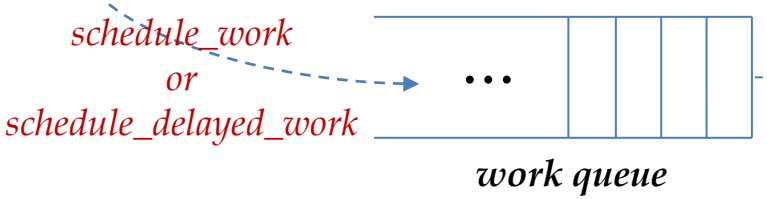
2. Top Half, Bottom Halves and Deferring Work - ***Work Queue(default)****

(*) work queue라고 하면, work, queue, worker thread의 세가지 요소를 통칭함.
(*) work queue를 위해서는 반드시 worker thread가 필요함.
(*) 기 정의된 worker thread(events/0)를 사용하는 방식과 새로운 worker thread 및 queue를 만드는 두 가지 방법이 존재함.
(*) work queue에서 사용하는 work는 sleep이 가능하다.
(*) worker thread 관련 보다 자세한 사항은 kernel/workqueue.c 파일 참조

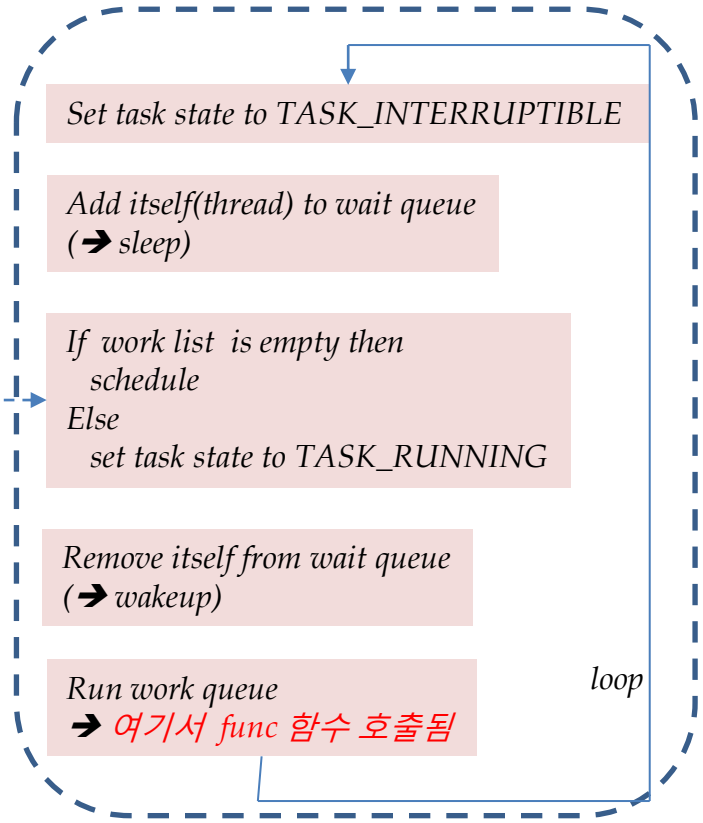
<my work – work_struct>




```
DECLARE_WORK(name,
                    void (*func)(void *));
or
INIT_WORK(struct work_struct *work,
                    void (*func)(void *));
```



<worker thread = events/0>



<기 정의된 work thread를 사용하는 경우>

schedule_work( )
➔ 엔트리를 work queue에 추가함
schedule_delayed_work( )
➔ 엔트리를 work queue에 추가하고, 처리를 지연시킴
flush_scheduled_work( )
➔ work queue의 모든 엔트리를 처리(비움). 모든 entry가 실행됨. Entry를 취소하는 것이 아님(주의). 또한 schedule_delayed_work은 flush시키지 못함.
cancel_delayed_work( )
➔ delayed work(엔트리)를 취소함.

2. Top Half, Bottom Halves and Deferring Work - *Work Queue(사용자 정의)**

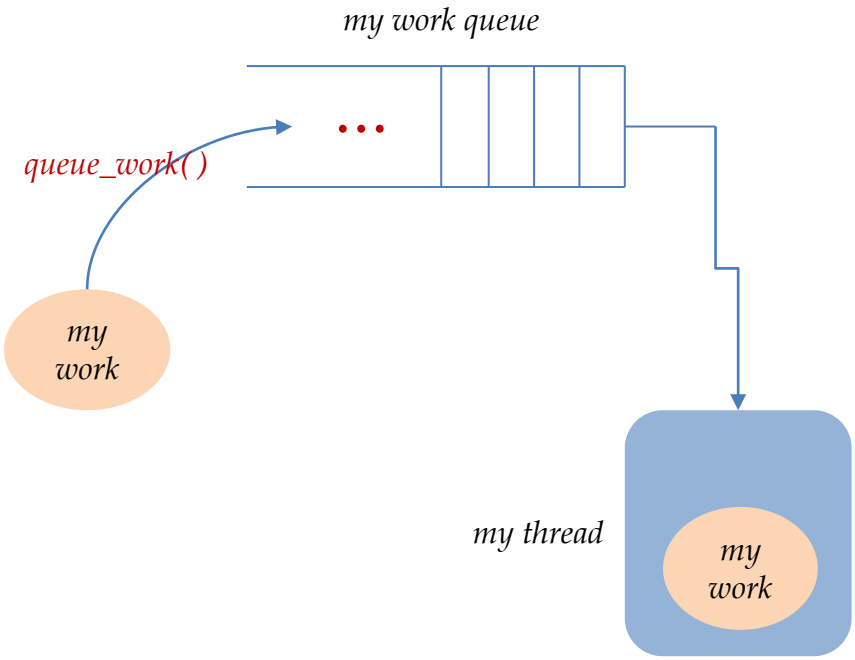
<사용자 정의 work queue 관련 API 모음>

*struct workqueue_struct **create_workqueue**(const char *name);*
➔ *사용자 정의 워크 큐 및 woker thread를 생성시켜 줌.*

*int **queue_work**(struct workqueue_struct *wq, struct work_struct *work);*
➔ *사용자 정의 work을 사용자 정의 work queue에 넣고, schedule 요청함.*

*void **flush_workqueue**(struct workqueue_struct *wq);*
➔ *사용자 정의 work queue에 있는 모든 work을 처리하여, queue를 비우도록 요청*

Delayed work 관련 API는 다음 페이지 참조 ➔



(create_workqueue에 인수로 넘겨준 name 값이 thread name이 됨 – ps 명령으로 확인 가능)

(*) 사용자 정의 work queue를 생성하기 위해서는 create_workqueue()를 호출하여야 하며, queue_work() 함수를 사용하여 work을 queue에 추가해 주어야 한다.
(*) 보통은 기 정의된 work queue를 많이 활용하나, 이는 시스템의 많은 driver 들이 공동으로 사용하고 있으므로, 경우에 따라서는 원하는 결과(성능)를 얻지 못할 수도 있다. 따라서 이러한 경우에는 자신만의 독자적인 work queue를 만드는 것도 고려해 보아야 한다.
(*) 보다 자세한 사항은 include/linux/workqueue.h 파일 참조

2. Top Half, Bottom Halves and Deferring Work - *Work Queue(사용자 정의)**

<Delayed work queue 관련 API 모음>

```
struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
};
```
➔ work과 timer를 묶어 새로운 data structure 정의 !

```
int schedule_delayed_work(struct delayed_work *work,
                          unsigned long delay);
```
➔ 주어진 delay 값 만큼 해당 work을 지연시켜 실행

```
int cancel_delayed_work(struct  delayed_work *work);
```
➔ 앞서 설명한 schedule_delayed_work으로 선언한 work을 중지(취소)

```
void flush_delayed_work(struct delayed_work *work);
```
➔ 사용자 정의 work queue에 있는 모든 delayed work을 처리하여, queue를 비우도록 요청

(*) __create_workqueue() 함수의 argument 값에 따라 4가지의 macro가 존재함 !!!
➔ 자세한 사항은 workqueue.h 파일 참조

예) mmc driver에서 발췌한 루틴

```
static struct workqueue_struct *workqueue; //선언

{
    …
    queue_delayed_work(workqueue, work, delay);
    // delayed work 요청
}


{
    …
    flush_workqueue(workqueue);
    // work queue에 있는 모든 flush 요청(delayed work에 대한 flush 아님)
}

{
    …
    workqueue = create_freezeable_workqueue("kmmcd");
    // work queue 생성
    …
    destroy_workqueue(workqueue);
    // work queue 제거
}
```

2. Top Half, Bottom Halves and Deferring Work - ***Kernel Threads****

Kernel thread란 ?

(*) kernel 내에서 background 작업(task)을 수행하는 목적으로 만들어진 lightweight process로, user process와 유사하나 kernel space에만 머물러 있으며, kernel 함수와 data structure를 사용하고, user space address를 포함하지 않는다(task_strruct 내의 mm pointer가 NULL임).

(*) 그러나, kernel thread는 user process와 마찬가지로 schedule 가능하며, 다른 kernel thread 혹은 interrupt handler 등에 의해 선점(preemtable)될 수 있다. 단, user-process에 의해 선점되지는 않음.

(*) 사용자 정의 kernel thread는 kthreadd(parent of kernel threads)에 의해 추가 생성(fork)된다.

kthread_create(my_thread)
or
kthread_run(my_thread)
➔ macro 임

kthread_create_list에 자신을 추가

```
int kthreadd(void *unused)
{
    …
    for (;;) {
      /* 생성할 thread가 없으면, 휴식 */
      /* 있으면, 아래 루틴 수행 */

      while (kthread_create_list is not empty)
        create_kthread(new_thread);

    }
}
```

2. Top Half, Bottom Halves and Deferring Work - ***Kernel Threads****

```
static void create_kthread(struct kthread_create_info *create)
{
    int pid;

    pid = kernel_thread(kthread, create,
                        CLONE_FS | CLONE_FILES | SIGCHLD);
    …
}
```

```
static int kthread(void *_create)
{
    struct kthread_create_info *create = _create;
    int (*threadfn)(void *data) = create->threadfn;
    void *data = create->data;

    …
    ret = threadfn(data);

    do_exit(ret);
}
```

사용자가 등록한 thread function (my_thread)수행 !

2. Top Half, Bottom Halves and Deferring Work - ***Kernel Threads****

*struct task_struct *kthread_create(**my_thread**, data, …);*

➔ *Kernel thread 생성(구동은 안함)*

➔ *보통은 **kthread_run**()을 더 많이 씀(thread 생성 후, 구동 시작)*

생성 및 구동

***kthread_stop**(tsk);*

구동 중지

int my_thread(void *data)

```
do {
  /* 특정 조건이 성립될 때까지, 대 기 ➔ 다른 코드에서 대기 조건을 해제 해주어야 함 (아래 코드는 단순 예임 */
  atomic_set(&cond, 0);
  wait_event_interruptible(wq,
      kthread_should_stop() ||
      atomic_read(&cond);

  /* 조건이 성립되면, 대기루틴을 나와, 실제 action 수행 */

  /* 실제 action 수행 부 */

} while (!kthread_should_stop());
```

kthread_run( )
➔ *kernel thread를 만들고, thread를 깨워줌*

kthread_create( )
➔ *kernel thread를 만듦(sleeping 상태로 있음)*

kthread_bind( )
➔ *thread를 특정 CPU에 bind 시킬 때 사용함.*

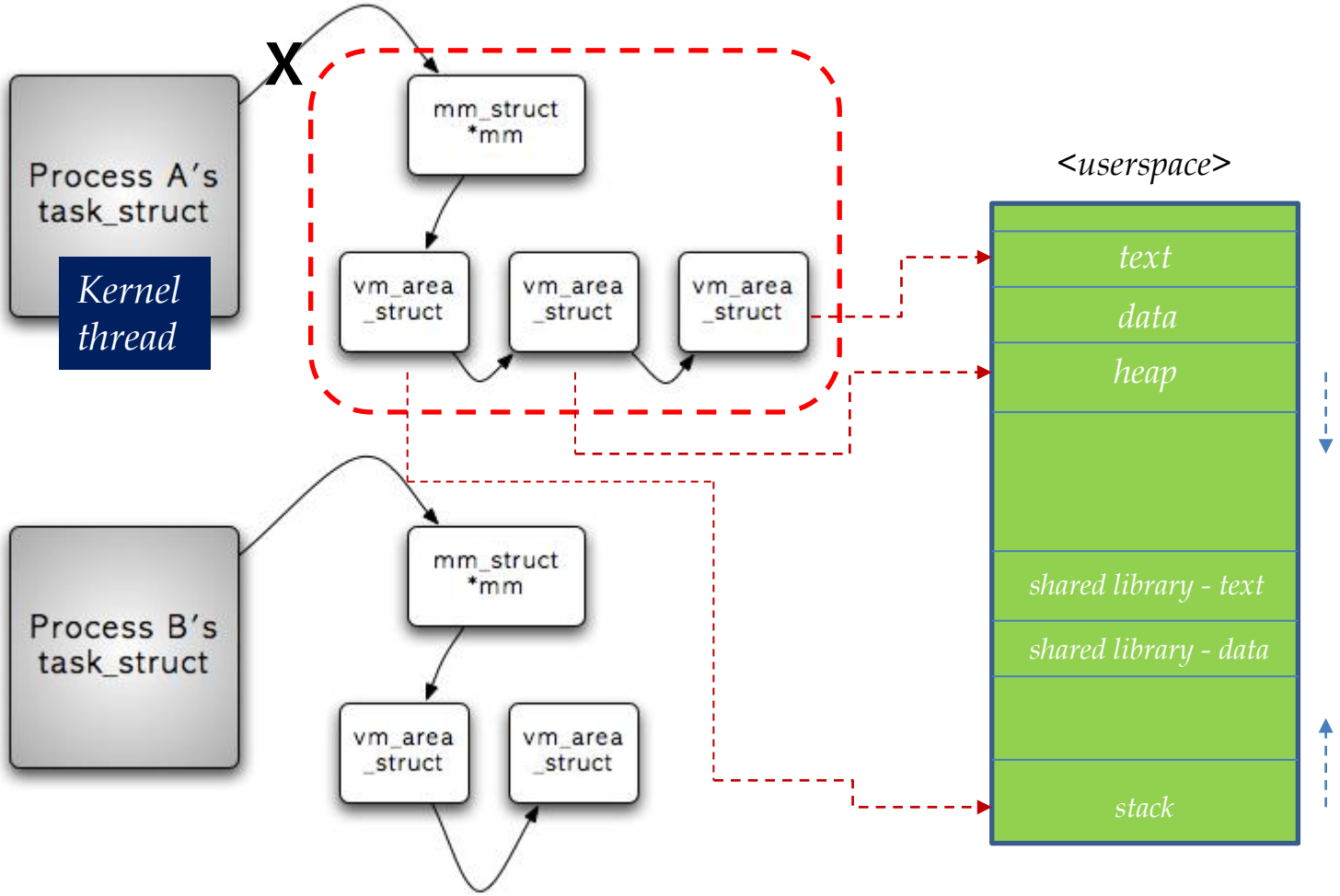
kthread_stop( )
➔ *thread를 중지할 때 사용함. Kthread_should_stoip을 위한 조건을 설정해 줌.*

kthread_should_stop( )
➔ *kernel thread 루틴을 멈추기 위한 조건 검사 함수.*

(*) work queue가 kernel thread를 기반으로 하고 있으므로, kernel thread를 직접 만들 필요 없이, Work queue를 이용하는 것이 보다 간편할 수 있다.
(*) 보다 자세한 사항은 include/linux/kthread.h 파일 참조

2. Top Half, Bottom Halves and Deferring Work – ***process(user context) & kernel thread의 차이***

(*) 아래 그림은 프로세스에 대한 memory 할당과 연관이 있는 mm_struct 및 vm_area_struct를 표현해 주고 있다.
(*) kernel thread는 user context 정보가 없는 process로 task_struct내의 mm field 값이 NULL 이다. 즉, 아래 그림에서 빨간색 점선 부분이 없다고 보면 된다.



다음 절로 넘어가기 전에 …*

Interrupt Latency의 원인

1) long-duration ISRs
2) Disabling interrupts
3) Prioritization of interrupts

Interrupt Thread

1) Interrupt는 kernel에게는 최대 latencey의 근원이다.
➔ 이를 줄이기 위해 kernel thread에거 일을 넘겨주고, 즉시 return.
➔ 이건 기존 bottom half와 차이가 없음.
➔ 차이점은, thread에게 우선 순위를 부여하여, preemptible thread context에서 동작하도록 함.

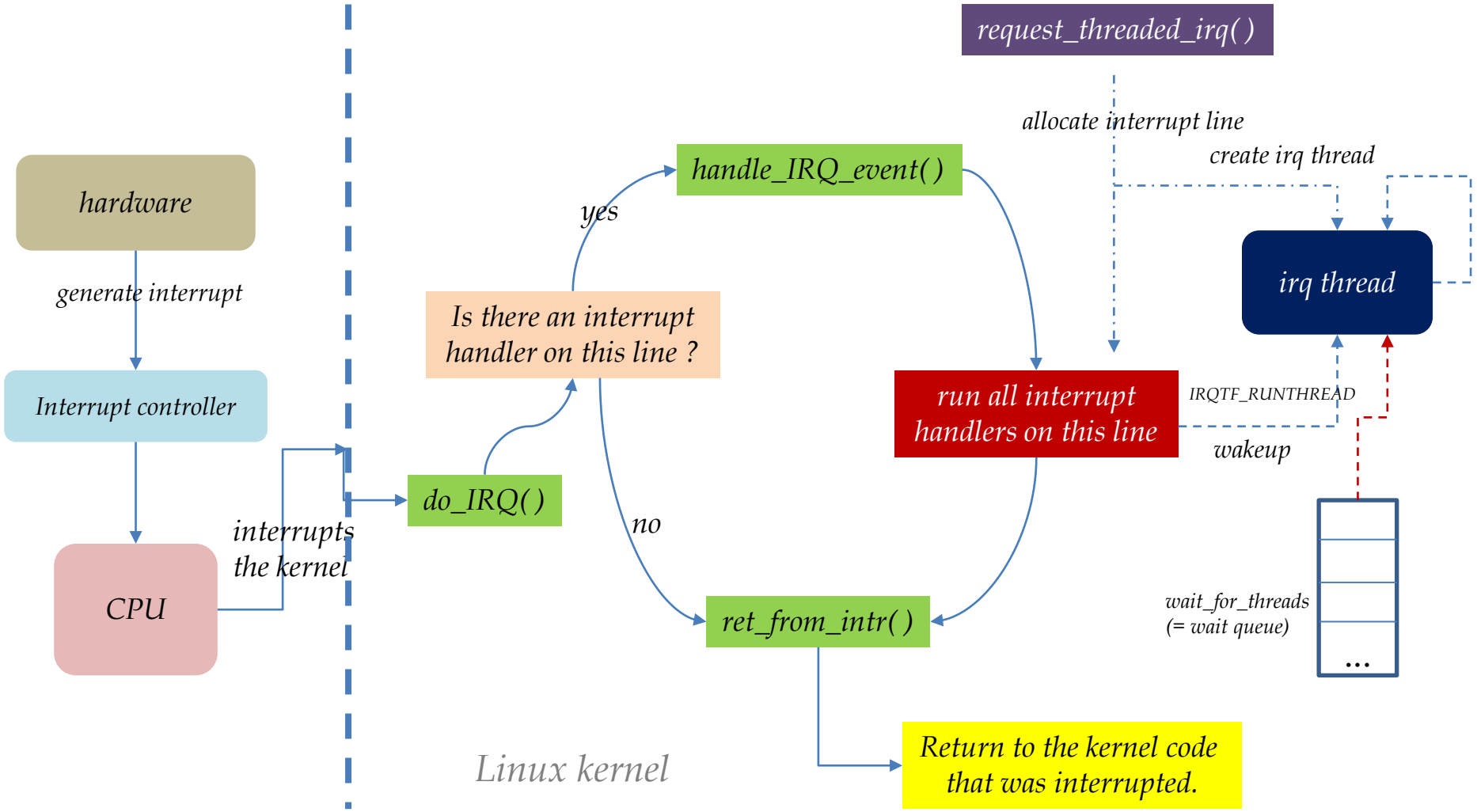
2) Bottom half 방식은 hard, soft handler간에 locking이 필요함.
➔ 이를 단순화 혹은 제거하여 복잡도 줄여야 함.

<work queue와 Interrupt thread의 차이점>

1) Work queue는 kernel thread이므로, scheduler가 선택하는 순간에 실행될 수 있다. 즉, 우선 순위가 높은 task가 실행 중이거나, 새로 들어오면 밀릴 수 있다. ➔ 이는 Real-Time OS에서 원하는 방식이 아님.

2) 반면에 Interrupt Thread는 우선 순위를 가지고 움직이므로, 즉시 실행될 수 있다. ➔ 단점은 interrupt 처리 면에서는 우수한 반응성을 보여주지만, 잦은 context switching은 전체 시스템의 성능 저하를 가져올 수도 있다.

2. Top Half, Bottom Halves and Deferring Work – ***Threaded Interrupt Handler****



(*) IRQ(line)당 1개씩의 kernel thread가 생성됨.
(*) 문제의 Shared IRQ의 경우는 위의 그림에서 처럼, handle_IRQ_event()가 모든 interrupt handler를 irq thread에게 순차적으로 던져주게 되며, irq thread가 이를 받아서 하나씩 처리하게 됨.
(*) 문제는 irq thread가 thread_fn을 처리하느라 바쁜 경우에는 어찌하느냐 인데 ...
➔ (당연히) thread에게 넘어간 task들이 CPU를 할당 못 받았으니 (irq thread용) wait queue에서 대기하게 되겠지 ...

2. Top Half, Bottom Halves and Deferring Work – ***Threaded Interrupt Handler****

HOT

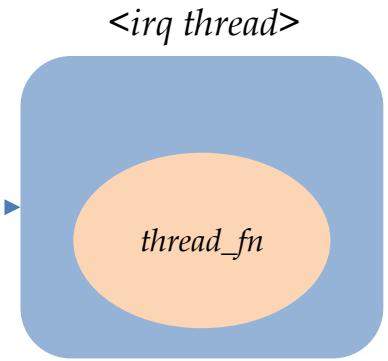
*int request_**threaded**_irq(unsigned int irq,*
irq_handler_t handler,
irq_handler_t thread_fn,
unsigned long flags,
*const char *name,*
*void *dev);*

➔ *Interrupt handler & threaded interrupt handler 등록 및 실행 요청*

➔ *Return 값: IRQ_NONE, IRQ_HANDLED, IRQ_WAKE_THREAD*

*(*) 이 방식은 hardware interrupt 방식과는 달리 Interrupt 요청 시, handler 함수를 kernel thread 에서 처리하므로, bottom half 방식으로 보아야 할 것임^^*

0) request_thread_irq() 호출시 irq thread 생성
1) If threaded interrupt comes, wakeup the irq thread.
2) Irq thread will run the <thread_fn>.



irq/number-name *형태로 thread명칭이 생성됨.*
(예: irq/11-myirq)

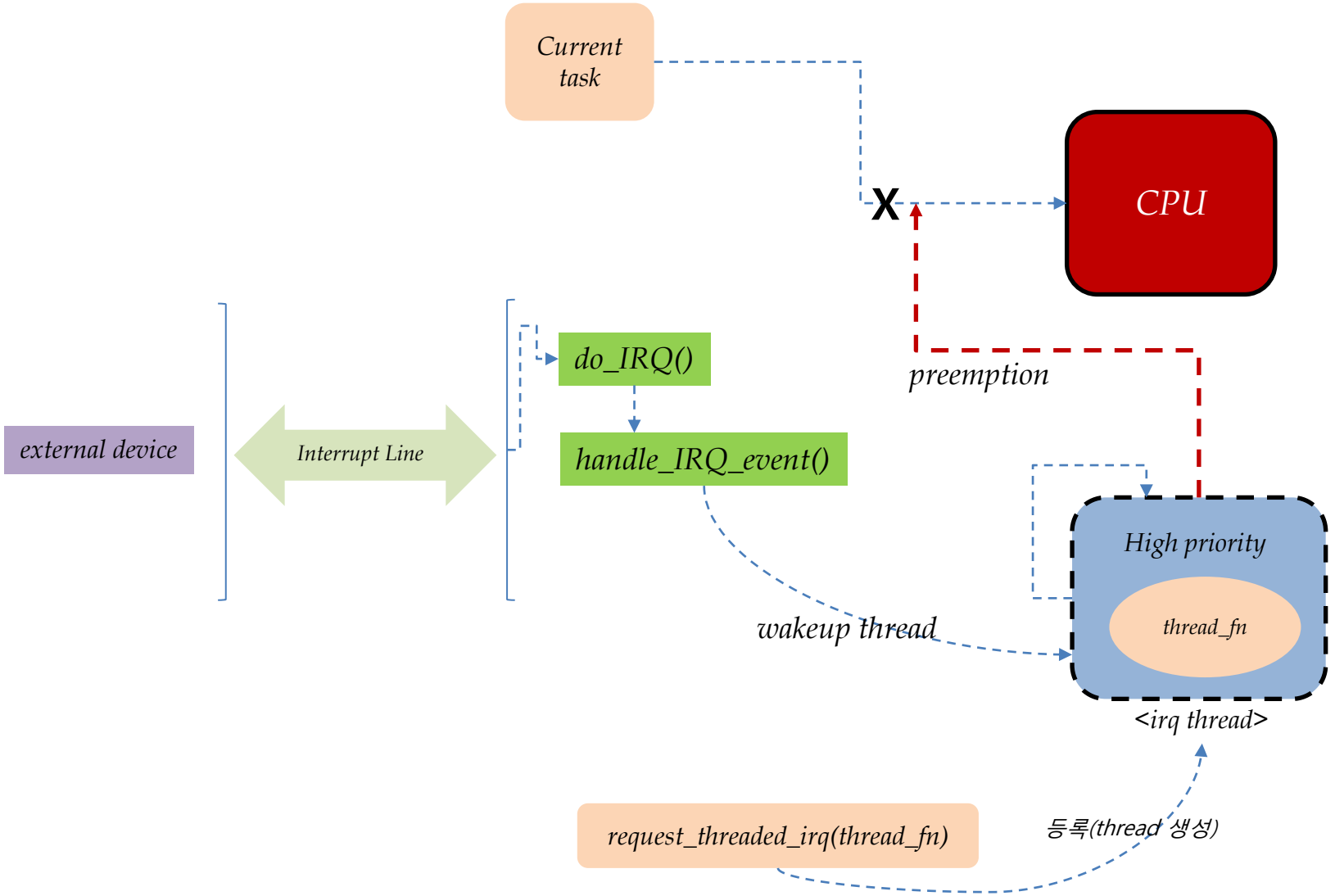
(*) 2.6.30 kernel 부터 소개된 기법(Real-time kernel tree에서 합류함)
➔ response time을 줄이기 위해, 우선 순위가 높은 interrupt 요청시 context switching이 일어남.
(*) interrupt 발생 시, hardware interrupt 방식으로 처리할지 Thread 방식으로 처리할지 결정(handler function)
➔ IRQ_WAKE_THREAD를 return하면, thread 방식으로 처리 (Handler thread를 깨우고, thread_fn을 실행함)
➔ 그렇지 않으면, 기존 hard interrupt handler로 동작함.
(*) handler가 NULL이고, thread_fn이 NULL이 아니면, 무조건 Threaded interrupt 방식으로 처리함.
(*) 이 방식은 앞서 소개한 tasklet 및 work queue의 존재를 위협할 수 있는 방식으로 인식되고 있음^^.
(*) 자세한 사항은 kernel/irq/handle.c, manage.c 파일 참조

2. Top Half, Bottom Halves and Deferring Work – *Threaded Interrupt Handler**

```
static int irq_thread(void *data)          /* ←IRQ 당 한개씩 할당되는 irq thread */
{
    struct irqaction *action = data;
    sched_setscheduler(current, SCHED_FIFO, MAX_USER_RT_PRIO/2);   /* 우선 순위 변경 */
    current->irqaction = action;

    while (1) {
        while (!kthread_should_stop()) {
            set_current_state(TASK_INTERRUPTIBLE);
            /* 아래 flag가 켜져 있으면, while loop을 빠져나와, irq thread function 수행 */
            /* IRQTF_RUNTHREAD는 handle_IRQ_event() 에서 설정해 줌 */
            if (test_and_clear_bit(IRQTF_RUNTHREAD, &action->thread_flags)) {
                __set_current_state(TASK_RUNNING);
                break;
            }
            schedule();      /* 할 일이 없으니, 휴식 */
        }
        raw_spin_lock_irq(&desc->lock);
        action->thread_fun(action->irq, action->dev_id);      /* requested_threaded_irq 에서 등록한
                                                                 thread_fun 함수 실행 – 실제 action */
        raw_spin_unlock_irq(&desc->lock);

        /* thread 에서 thread_fn 수행 중, 새로운 interrupt가 들어올 경우,
           wait queue 에 누적되고, 아래에서 이를 깨우는 듯 !!! */
        if (wait_queue_active(&desc->wait_for_threads)
            wake_up(&desc->wait_for_threads);                 /* wait queue 에 대기 중인 task를 깨움 */
    }
}
```

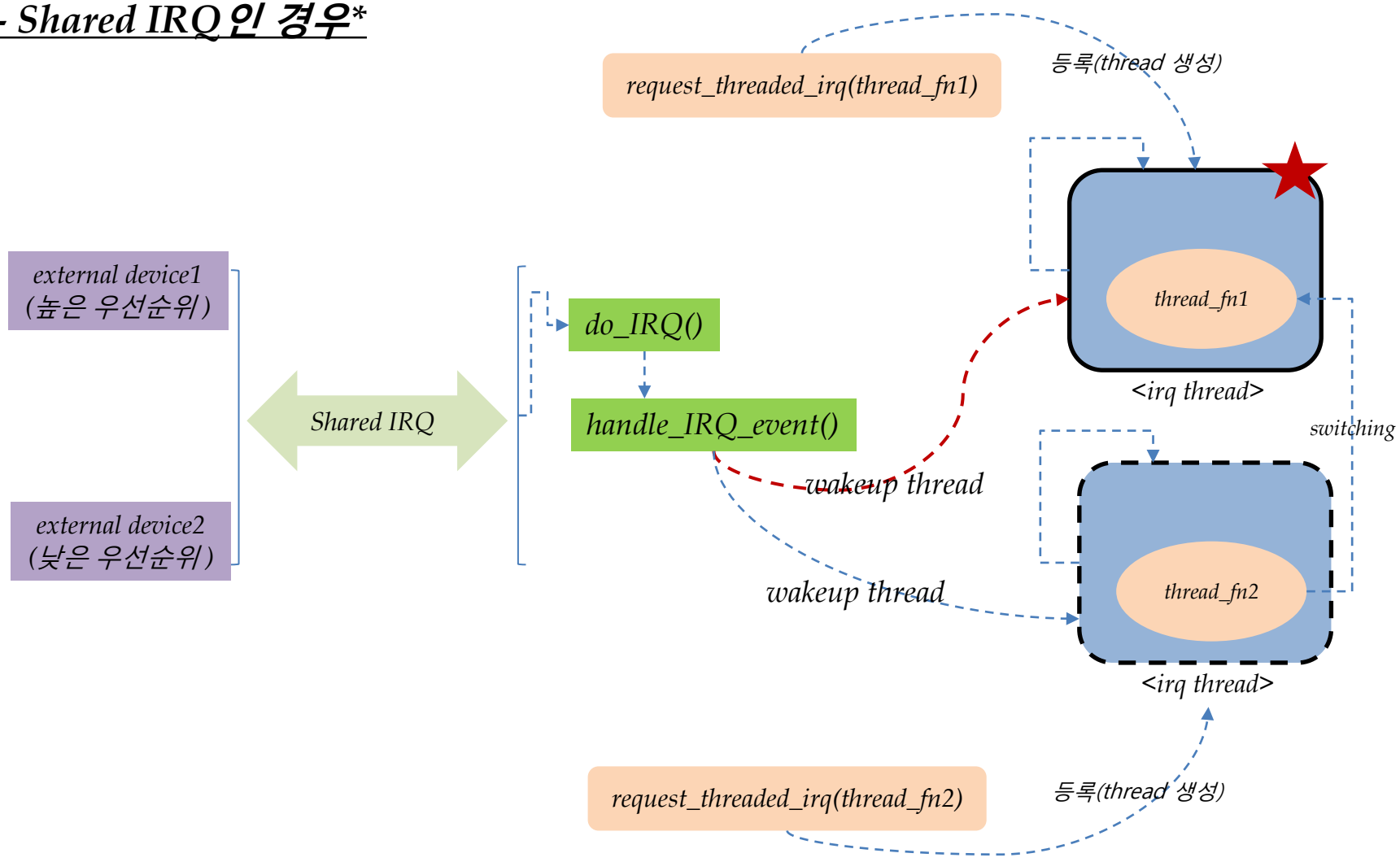
2. Top Half, Bottom Halves and Deferring Work – ***Threaded Interrupt Handler***

*



(*) work queue와는 달리, threaded interrupt handler는 높은 우선 순위로 동작하므로, CPU를 선점할 수 있다.

2. Top Half, Bottom Halves and Deferring Work – ***Threaded Interrupt Handler – Shared IRQ인 경우****



(*) work queue와는 달리, threaded interrupt handler를 사용하면, 우선 순위가 높은 놈(?)이 치고 들어올 경우, 이를 바로 처리(real-time)하는 것이 가능하다.
(*) 위의 그림에서 두개의 thread를 그렸으나, 실제로는 같은 하나의 thread임(IRQ 당 1개의 thread만 생성됨)

(*) Real-Time Patch 관련 참고 사항

R-T Patch 주요 사항

1) Spinlock이 PI-Mutex로 교체됨.
➔ 우선순위 상속 지원
➔ raw_spinlock이 old spinlock을 대신함.

2) spinlock_t와 rwlock_t로 보호되던 critical section을 이제는 선점(preemptible) 할 수 있게 됨.

3) Old linux timer API를 high-resolution timer와 normal timer로 분리함.

4) Interrupt handler를 preemptible thread context에서 돌리도록 함.
➔ hard/soft IRQ가 모두 thread context에서 돌아감.

2. Top Half, Bottom Halves and Deferring Work – *ps**

Kernel thread daemon
(모든 kernel thread의 parent)

Softirq/tasklet을 위한
Kernel thread

```
# ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root       1      0       348      212   800f19e8 0000877c S /init
root       2      0       0          0     8009021c 00000000 S kthreadd
root       3      2       0          0     8007f5d8 00000000 S ksoftirqd/0
root       4      2       0          0     8008ca44 00000000 S events/0
root       5      2       0          0     8008ca44 00000000 S khelper
```

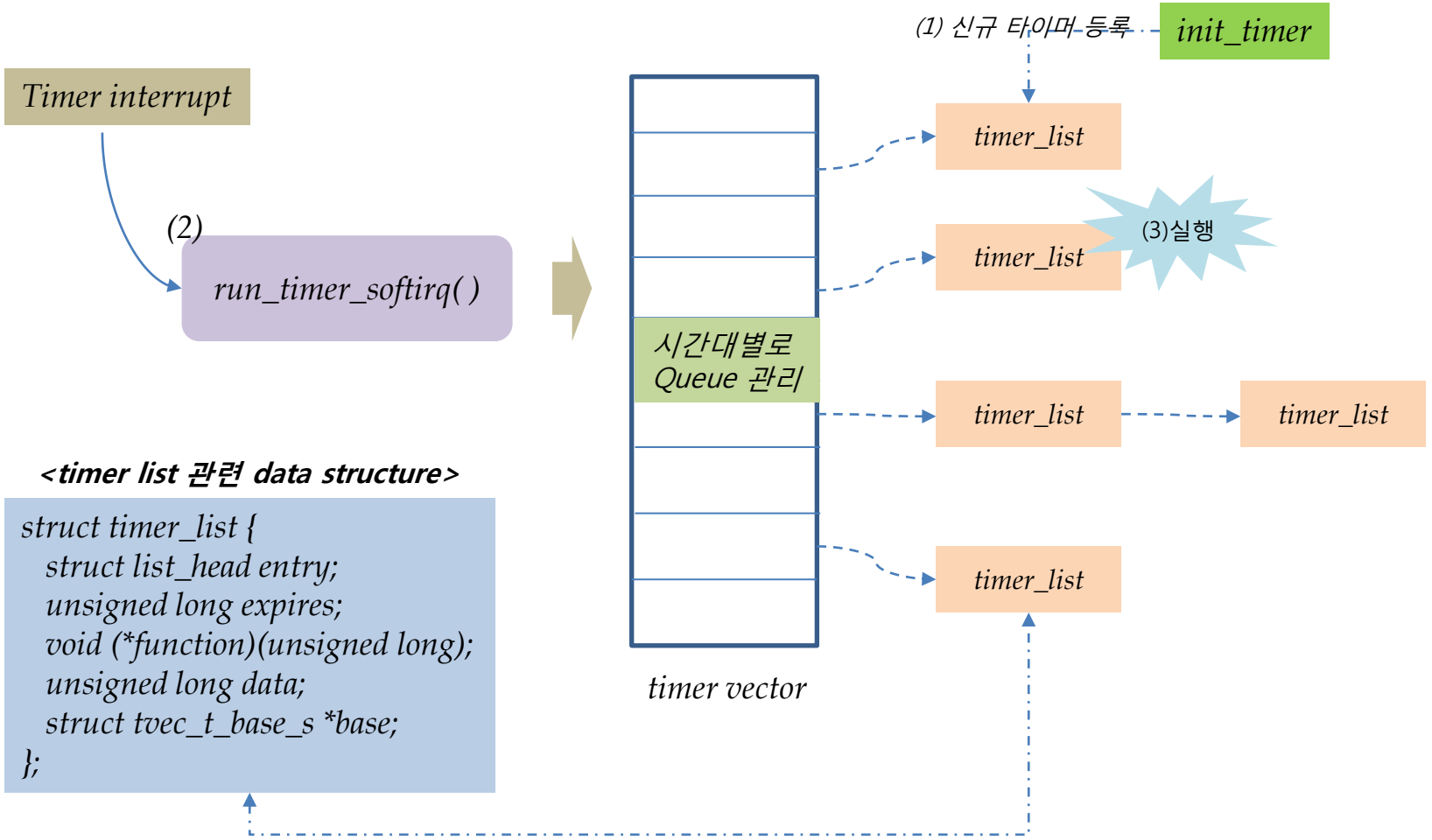
Work queue의 한 형태
(usermodehelper – kernel에서
User process 실행 – 역할 수행

Default work queue thread
(worker thread)

*(*) 위에는 표시되지 않았으나, 사용자 정의 work queue를 만들 경우 혹은 kernel thread를 생성할 경우, 자신 만의 work queue 혹은 kernel thread가 ps 결과로 보이게 될 것임^^*
(*) ksoftirqd/0와 events/0의 0은 첫 번째 processor를 의미함.

3. Timer(1)*

(*) 앞서 설명한 bottom half의 목적은 work을 단순히 delay시키는데 있는 것이 아니라, 지금 당장 work을 실행하지 않는데 있음. 한편 timer는 일정한 시간 만큼 work을 delay시키는데 목적이 있음 !
➔ Bottom half(threaded interrupt handler는 예외)의 경우는 delay 시간을 보장받기 힘들다^^.
(*) timer는 timer interrupt를 통해 동작하는 방식을 취함(software interrupt).
즉, 처리하려는 function을 준비한 후, 정해진 시간이 지나면 timer interrupt가 발생하여 해당 function을 처리하는 구조임.
(*) timer는 cyclic(무한 반복) 구조가 아니므로, time이 경과하면 timer function이 실행되고, 해당 timer는 제거된다.

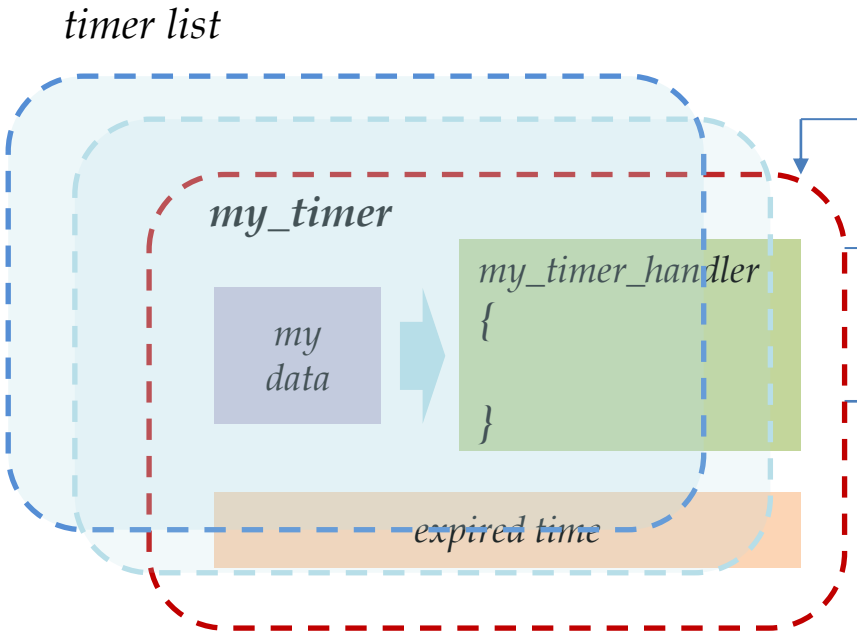


<timer list 관련 data structure>

```
struct timer_list {
  struct list_head entry;
  unsigned long expires;
  void (*function)(unsigned long);
  unsigned long data;
  struct tvec_t_base_s *base;
};
```

3. Timer(2)*

(*) timer를 deactivation 시키는 함수에는 del_timer()와 del_timer_sync()가 있음.
(*) del_timer_sync()는 현재 실행 중인 timer handler가 끝날때까지 기다려 준다.
따라서 대부분의 경우에 이 함수를 더 많이 사용한다. 단, 이 함수의 경우는 interrupt context에서는 사용이 불가하다.



(3) add_timer()
or
mod_timer( )
➔ 이를 호출하여 반복적으로 timer를 돌릴 수 있음
➔ mod_timer는 expiratoin 값을 변경함은 물론이고, Timer를 activation 시켜준다.

Time expired interrupt 시 호출됨

void init_timer(struct timer_list *timer);
or
TIMER_INITALIZER(_function, _expires, _data)
➔ (1)초기화

my_timer_handler(my_data) will be run !
➔ 얘는 빠르게 처리되는 코드이어야 함 !
➔ Softirq와 같은 bottom half conntext 임

my_timer.expires = jiffies + delay;
my_timer.data = my_data;
my_timer.function = my_timer_handler;
➔ (2) 설정

3. Timer(3) – ***HZ & Jiffies****

HZ: *the frequency of system timer(= tick rate)*
➔ *초당 가능한 tick의 수(= 주파수 개념)*
➔ *CPU(성능) 마다 값이 다름.*

Jiffies: *the number of ticks that have occurred since the system booted.*
➔ *시스템이 부팅 한 이후로 발생한 tick의 수*

Seconds * HZ = jiffies
➔ *초(seconds)를 이용하여 jiffie 값 구하기*

(jiffies / HZ) = seconds
➔ *Jiffie 값으로 부터 초 계산하기*

예)

```
unsigned long time_stamp = jiffies;   /* 현재 */
unsigned long later = jiffies + 5*HZ;  /* 지금 부터 5초 후 */
unsigned long fraction = jiffies + HZ/10;
unsigned long next_tick = jiffies + 1; /* 지금 부터 1 tick 후 */
```

3. Timer(4)*

schedule_timeout (timeout): *현재 실행 중인 task에 대해 delay를 줄 수 있는 보다 효과적인 방법. 이 방법을 사용하면 현재 실행 중인 task를 지정된 시간이 경과할 때까지 sleep 상태(wait queue에 넣어 줌)로 만들어 주고, 시간 경과 후에는 다시 runqueue에 가져다 놓게 함. schedule_timeout( )의 내부는 timer와 schedule 함수로 구성되어 있음.*

```
schedule_timeout (signed long timeout)
{
    timer_t timer;
    unsigned long expire;

    ...
    expire = timeout * jiffies;

    init_timer(&timer);
    timer.expires = expire;
    timer.data = (unsigned long)current;
    timer.function = process_timeout;

    add_timer(&timer);
    schedule();
    del_timer_sync(&timer);

    timeout = expire – jiffies;
    ...
}
```

(*) schedule_timeout 말고도, process scheduling과 조합한 타이머 리스트 관련 함수로는 아래와 같은 것들이 있다.

➔ ***process_timeout(), sleep_on_timeout(), interruptible_sleep_on_timeout()***

3. Timer(5) – ***msleep****

```
void msleep(unsigned int msecs)
{
        unsigned long timeout = msecs_to_jiffies(msecs) + 1;

        while (timeout)
            timeout = schedule_timeout_uninterruptible(timeout);
}
```

(*) msleep도 나쁘지 않군요. 내부적으로는 schedule_timeout function을 쓰네요. 다만 차이점이 있다면, uninterruptible로 되어 있어, 주어진 시간 만큼은 확실이 sleep 상태에 있겠네요.

(*) 참고로, 이와 유사한 msleep_interruptible( )을 쓰면, sleep하고 있다가, wakeup 조건 (다른 task들이 놀고 있어, 내게 차례가 올 경우)이 될 경우, 주어진 시간을 다 채우지 않은 상태에서도 깨어날 수 있습니다^^.

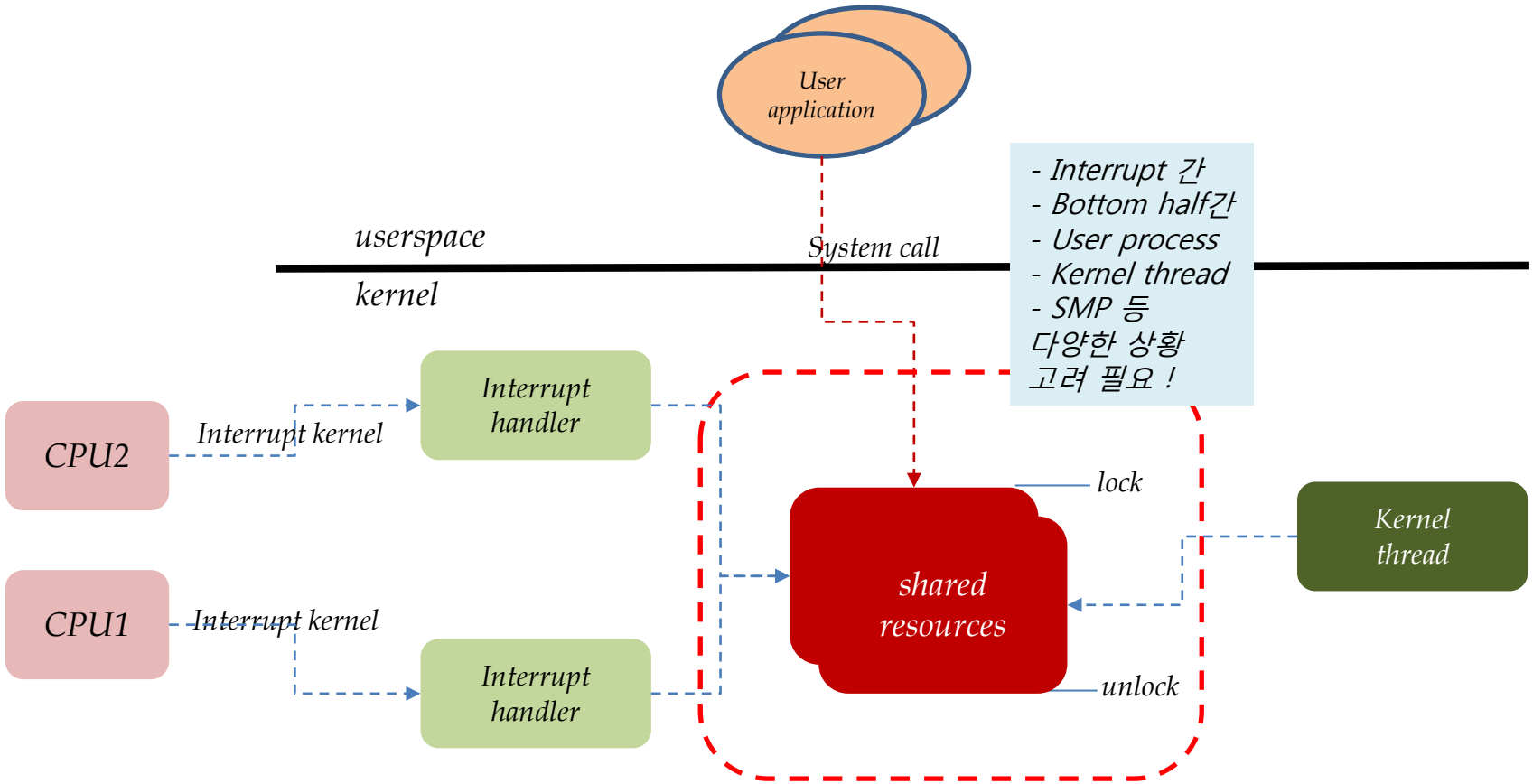
3. Timer(6) – ***High Resolution Timers***

(*) hrtimer는 기존 HZ 단위의 low-resolution timer가 mili-초 수준의 정밀도를 제공하는 것과는 달리 nano 초 단위로 시간을 관리하는 방식이다.
(*) 당연한 얘기겠지만, 세밀한 단위로 시간(clock)을 제공해 주는 장치가 시스템 내에 존재해야 한다.
(*) hrtimer의 핵심은 항상 동일한 주기로 계속 timer interrupt가 발생하는 것이 아니라, 특정 event가 일어날 시점을 정확히 지정하여 timer를 등록하고, 해당 시점에 timer interrupt가 발생하면, 그 때 event를 처리하는 것이다. 따라서 발생할 event의 유무 및 간격에 따라 timer interrupt의 주기가 바뀌게 된다.

```
static struct kt_data {
    struct hrtimer timer;
    ktime_t period;
} *data;

static enum hrtimer_restart ktfun(struct hrtimer
*var)
{
    ktime_t now = var->base->get_time();
    /* … */
    hrtimer_forward(var, now, data->period);
    return HRTIMER_NORESTART;
}

static void __exit my_exit(void) {
    hrtimer_cancel(&data->timer);
    kfree(data);
}
```

```
static init __init my_init(void)
{
    data = kmalloc(sizeof(*data), GFP_KERNEL);

    data->period = ktime_set(1, 0); /* nano 초 지정 가능 */
    hrtimer_init(&data->timer,
      CLOCK_REALTIME, HRTIMER_MODE_REL);
    data->timer.function = ktfun;
    hrtimer_start(&data->timer, data->period,
                  HRTIMER_MODE_REL);
    return 0;
}
```

4. Synchronization(1) – *Concurrency(& Pseudo concurrency) 상황**



(*) system call은 임의의 시점에서 발생할 수 있다(Pseudo-concurrency)
(*) interrupt도 임의의 시점(asynchronously)에 발생할 수 있으며, CPU가 두 개 이상일 경우, 각각의 CPU로 부터 동시에 서로 다른 임의의 Interrupt가 발생할 수 있다(Concurrency).
➔ Linux kernel은 fully preemptive한 특성을 가지고 있으므로, 각각의 경우에 kernel code에서 shared data를 사용(race condition)하고 있다면, 처리에 신중(locking)을 기해야 할 것임.

4. Synchronization(2) – *Concurrency(& Pseudo concurrency) 상황**

(*) 아래와 같은 concurrency 상황이 발생할 수 있으며, 동시에 실행 가능한 상황에 처해 있는 코드는 적절히 보호되어야 한다.

***Interrupts**: interrupt는 아무 때나 발생(asynchronously)하여, 현재 실행중인 코드를 중단시킬 수 있다.*

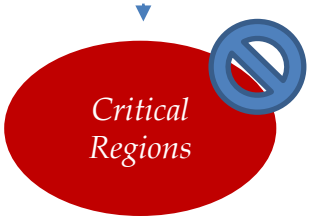
***Softirqs & tasklet**: 얘들은 kernel이 발생시키고, schedule하게 되는데, 얘들도 거의 아무때나 발생하여 현재 실행중인 코드를 중단시킬 수 있다.*

***Kernel preemption**: 글자 그대로 한 개의 task가 사용하던 CPU를 다른 task가 선점(CPU를 차지)할 수 있다(**linux 2.6 부터는 fully preemptive**).*

***Sleeping and synchronization with user-space**: kernel task는 sleep할 수 있으며, 그 사이 user-process(system call)가 CPU를 차지할 수 있다.*

***Symmetrical multiprocessing**: 두 개 이상의 processor(CPU)가 동시에 같은 kernel code를 실행할 수 있다.*

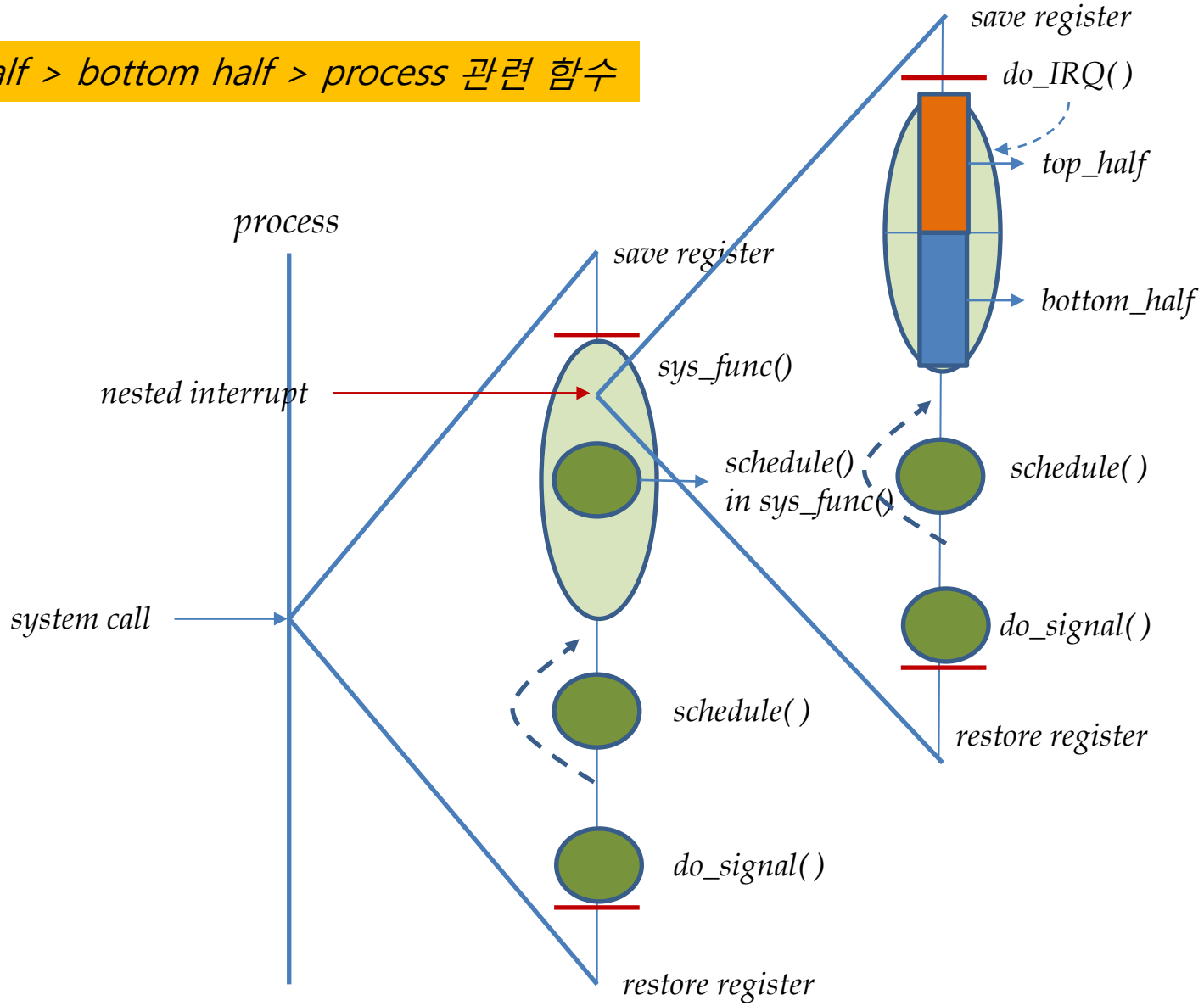
1) SMP
2) Interrupt handlers
3) Preempt-kernel
4) Blocking methods



<Kernel preemption이 발생하는 경우>
1) Interrupt handler가 끝나고, kernel space로 돌아갈 때
2) Kernel code가 다시 preemptible해 질 때(코드 상에서)
3) Kernel task가 schedule( ) 함수를 호출할 때
4) Kernel task가 block될 때(결국은 schedule( ) 함수를 호출하는 결과 초래)

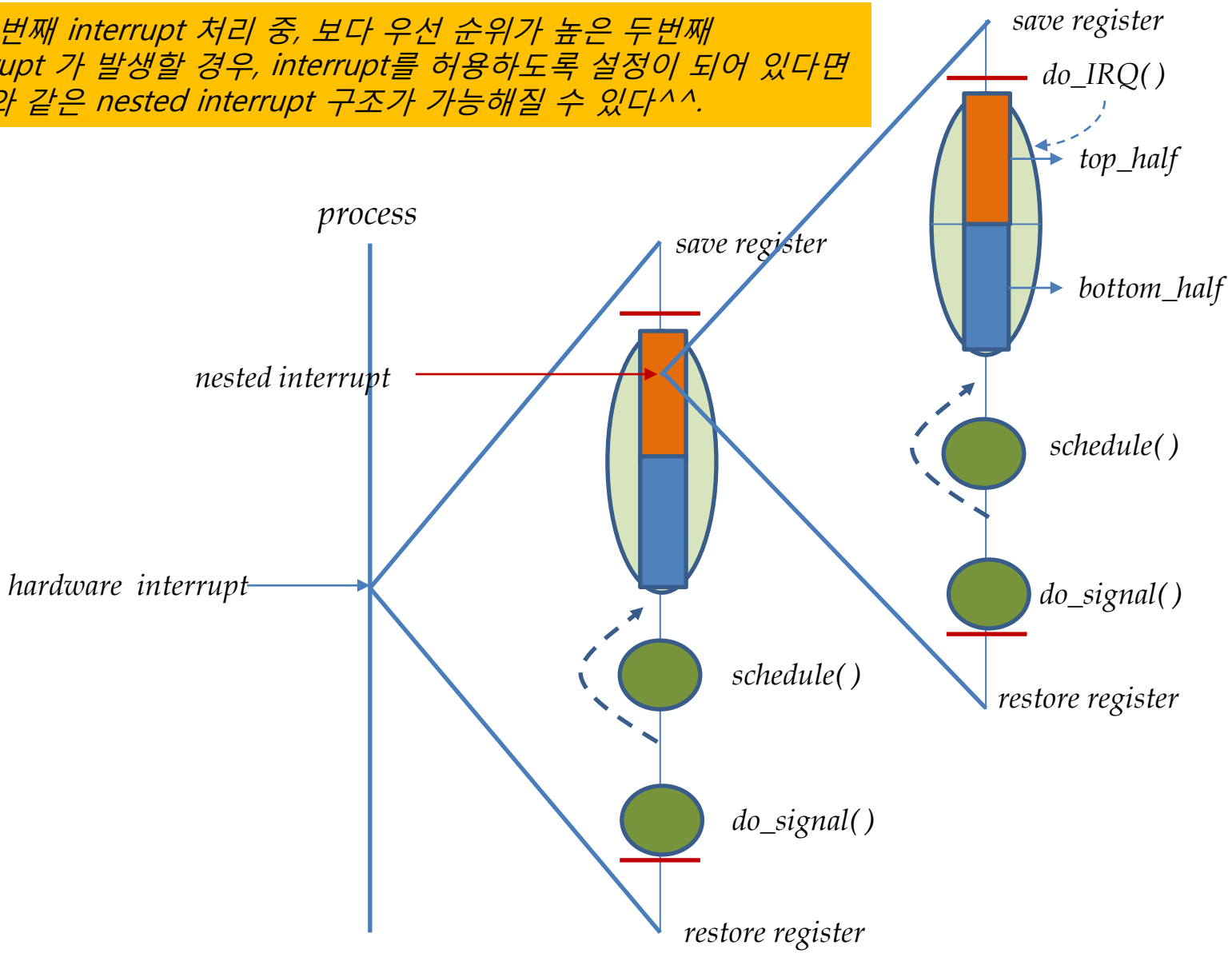
4. Synchronization(3) – ***kernel preemption/1***

(*) top half > bottom half > process 관련 함수



4. Synchronization(3) – ***kernel preemption/2***

(*) 첫번째 interrupt 처리 중, 보다 우선 순위가 높은 두번째 interrupt 가 발생할 경우, interrupt를 허용하도록 설정이 되어 있다면 아래와 같은 nested interrupt 구조가 가능해질 수 있다^^.



4. Synchronization(4) – *Coding시 주의 사항/1**

Thread 1

Try to lock the queue
Succeeded: acquired lock
Access queue…
Unlock the queue
…

Thread 2

Try to lock the queue
failed: waiting…
Waiting …
Waiting …
Succeeded: acquired lock
Access queue…
Unlock the queue
…

1) *Global data 인가? 즉, 여기 말고 다른 곳(thread of execution) 에서도 이 data 에 접근이 가능한가 ?*

2) *Process context 와 interrupt context 에서 공유가 가능한 data 인가 ?*

3) *아니면, 두 개의 서로 다른 interrupt handler 에서 공유가 가능한 data 인가 ?*

4) *Data 를 사용하던 중에 다른 process 에게 CPU 를 뺏길 경우, CPU 를 선점한 process 가 그 data 를 access 하지는 않는가 ?*

5) *현재 process 가 sleep 하거나 block 될 수 있는가 ? 만일 그렇다면, 이때 사용중이던 data 를 어떤 상태로 내버려 두었는가 ?*

6) *내가 사용 중이던 data 를 해제하려고 하는데, 이를 누군가가 막고 있지는 않은가 (사용할 수 있지는 않은가) ?*

7) *이 함수를 시스템의 다른 processor(CPU) 에서 다시 호출한다면 어떻게 되는가 ?*

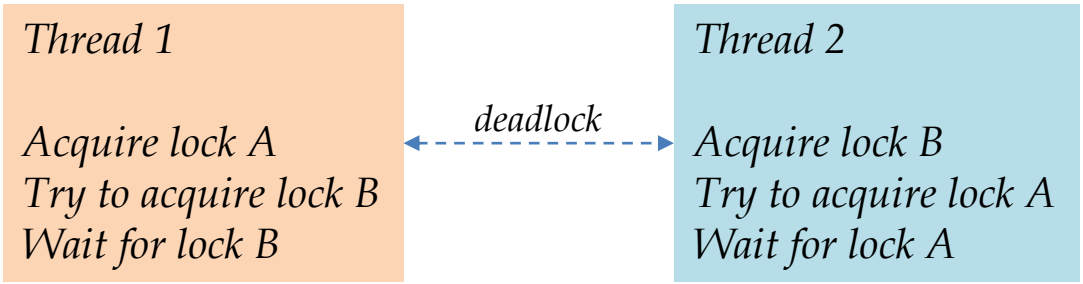
8) *내가 짠 code 가 concurrency 상황에 안전하다고 확신할 수 있는가 ?*

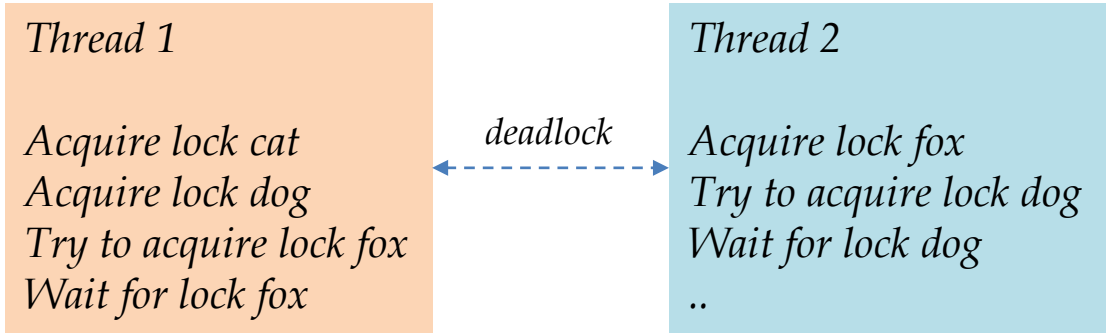
4. Synchronization(5) – *Coding시 주의 사항/2**

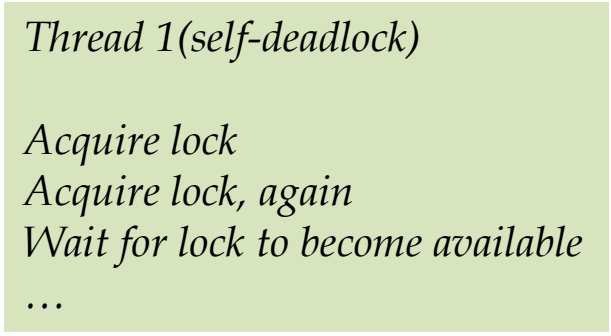
	<Case 1> A) process context B) Tasklet	<Case 2> A) Tasklet B) Tasklet	<Case 3> A) Softirq B) interrupt	<Case 4> A) Interrupt B) Interrupt	<Case 5> A) Work queue B) Kernel thread	<case 6> A) Kernel thread B) Interrupt	비고
A가 B에 의해 선점될 수 있는가 ?	yes	no	yes	yes	yes	yes	local_irq_disable 사용해야함. spin_lock_bh (bottom half 간) mutex (process context 보호 시)
A의 critical section이 다른 CPU에 의해 접근될 수 있는가 ?	yes	yes	yes	yes	yes	yes	spin_lock 사용해야 함.

1) Interrupt handler가 실행 중일 때, 다른 interrupt handler들이 저절로 block되는 것은 아니다.
➔ 단, 같은 interrupt line은 block을 시킴.
2) 한 CPU의 interrupt가 disable되었다고, 다른 CPU의 interrupt가 disable되는 것은 아니다.
3) softirq(tasklett)는 다른 softirq를 선점하지는 않는다.

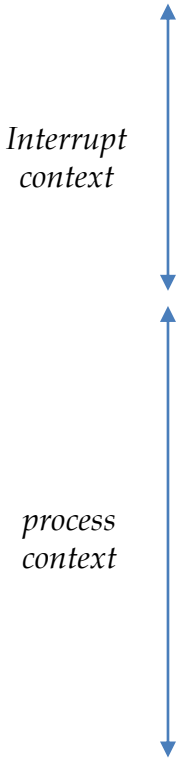
4. Synchronization(6) – *Coding시 주의 사항/3**







4. Synchronization(7) – *Sync. Methods**



Kernel Synchronization Methods	내용 요약/특징
Atomic operations	
Spin Locks	Low overhead locking Short lock hold time Need to lock from interrupt context
Reader-Writer Spin Locks	
Semaphores	
Reader-Writer Semaphores	
Mutexes	Long lock hold time Need to sleep while holding lock
Completion Variables	
BKL(Big Kernel Lock)	
Sequential Locks	
Preemption Disabling	Preempt_disable()/preempt_enable() ➔ Kernel preemption을 금지/허용
Barriers	Instruction reordering 금지 명령 (spinlock, irq disable, preempt_disable 등의 내부를 구성)

4. Synchronization(8) – ***Sync. Methods****

Atomic Operations
(수행 중에는 interrupt 등에 의해 중단되지 않음)

```
atomic_t v = ATOMIC_INIT(0);
atomic_set(&v, 4);   /* v = 4 (atomically) */
atomic_add(2, &v);  /* v = v + 2 = 6 (atomically) */
atmic_inc(&v);       /* v = v + 1 = 7 (atomically) */
…
```

Interrupt context

Spinlocks
(resource를 사용할 수 있을 때까지, sleep하지 않고 기다림 – spin)

```
spinlock_t my_lock;
spin_lock_init(&my_lock);

spin_lock(&my_lock);
…. Critical region …
spin_unlock(&my_lock);
```

***Spinlocks** with interrupt-disabling*
(resource를 사용할 수 있을 때까지, sleep하지 않고 기다림. 동시에 interrupt를 금지시킴

```
spinlock_t my_lock;
unsigned long flags;
spin_lock_init(&my_lock), flags;

spin_lock_irqsave(&my_lock);
…. Critical region …
spin_unlock_irqrestore(&my_lock, flags);
```

(*) spinlock은 SMP환경에서 사용되도록 만들어 졌으나, 선점형 kernel인 2.6에서는 Uniprocessor 환경에서도 필요하다.

4. Synchronization(9) – ***Sync. Methods****

(*) 한 thread가 lock을 얻은 경우, 그 다음에 진입하는 Thread는 wait queue에서 대기(sleep)하게 된다.

Semaphores

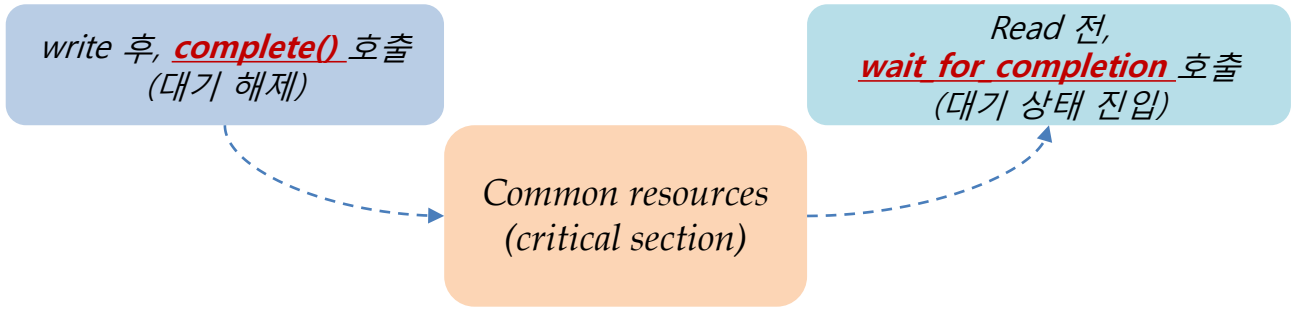
```
static DECLARE_MUTEX(mr_sem);

if (down_interruptible(&mr_sem)) {
   /* signal received, semaphore not acquired. */
}

/* critical region ... */

up(&mr_sem);
```

Process context

Mutexes

```
DEFINE_MUTEX(name); /* or mutex_init(&mutex) */

mutex_lock(&mutex);

/* critical region ... */

mutex_unlock(&mutex);
```

4. Synchronization(10) - ***Completion****

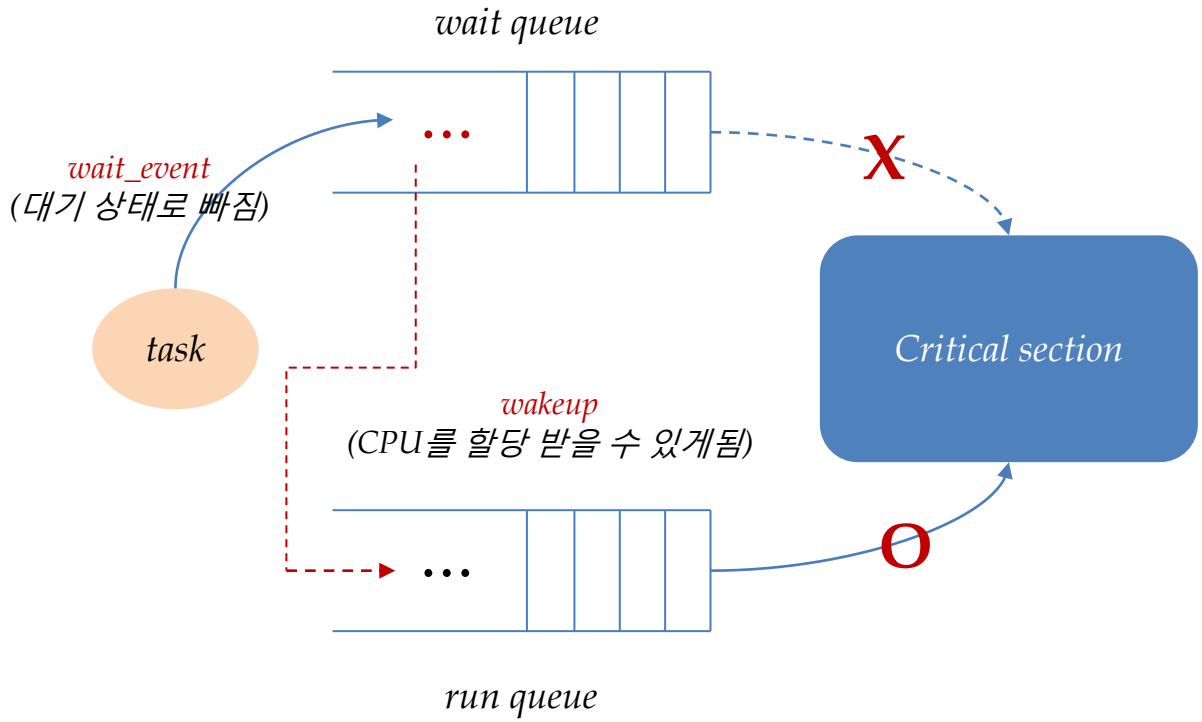


(*) 얘는 앞 페이지에 제시한 방법과는 약간 다르게, 서로 다른 두 개의 코드간에 동기(순서 부여)를 맞추고자 할 때 매우 유용하게 사용될 수 있다.

```
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};

void init_completion(struct completion *c);   /* DECLARE_COMPLETION(x)도 사용 가능 */
  ➔ completion 초기화
void wait_for_completion(struct completion *c); /* timeout 함수도 있음 */
  ➔ critical section에 들어갈 때 호출(대기를 의미함)
int wait_for_completion_interruptible(struct completion *c); /* timeout 함수도 있음 */
   ➔ critical section에 들어갈때 호출(대기를 의미함). 이 함수 호출 동안에 Interrupt 가능함.

void complete(struct completion *c);
   ➔ critical section에 들어 갈 수 있도록 해줌(대기 조건을 해지해 줌)
void complete_and_exit(struct completion *c, long code);
```

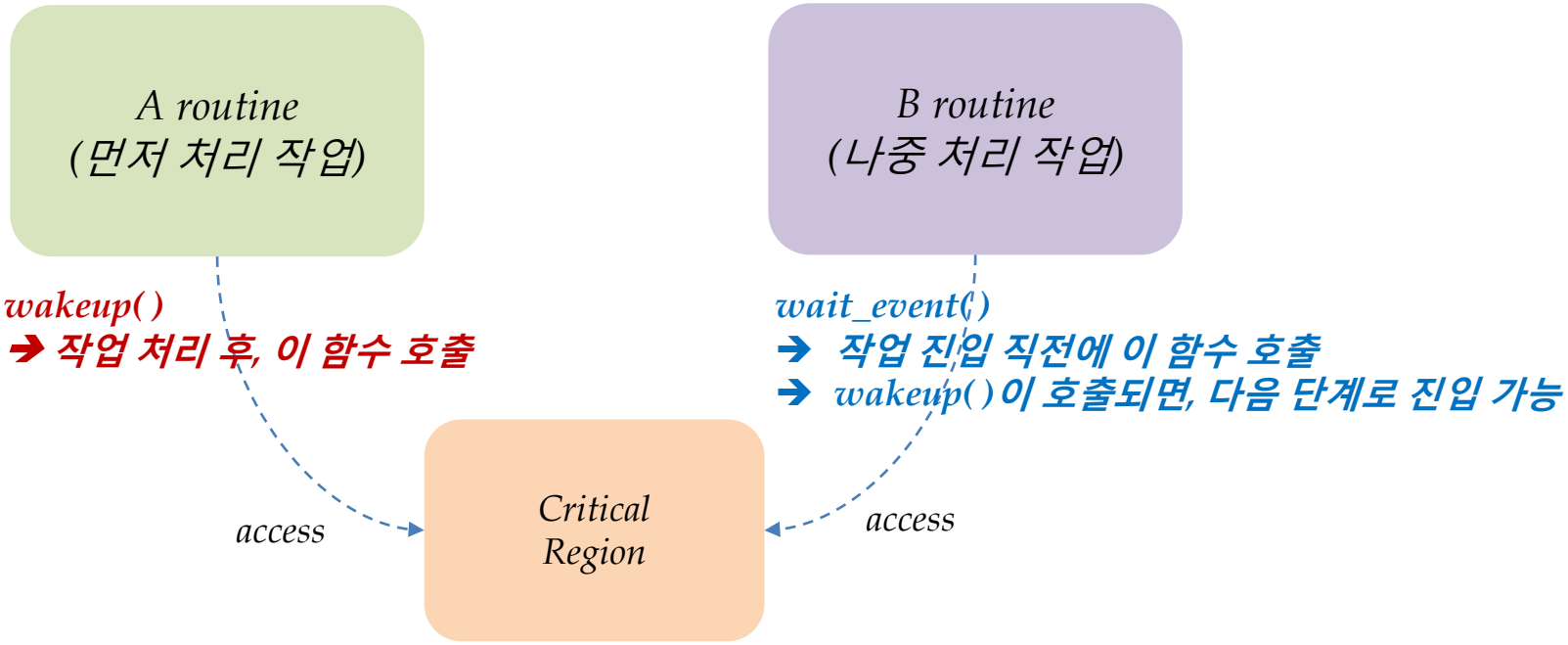
4. Synchronization(11) - ***Sleeping & Wait Queue****

(*) 앞 페이지의 synchronization method에는 없었으나, wait queue를 이용하여도 동일한 효과를 얻을 수 있다^^.
(*) 앞서 설명한 completion의 내부는 wait queue로 구현되어 있음.



4. Synchronization(12) - *Sleeping & Wait Queue**

(*) 아래 그림은 앞 페이지의 그림을 race condition 관점에서 다시 그린 것이다.
(*) 동일한 resource를 두 개의 서로 다른 code가 access할 수 있는 상황에서, <A routine>에게 높은 우선순위를 부여하고자 할 경우에는, 아래와 같이 <B routine>은 wait_event( )함수를 호출하여 대기 상태로 진입해야 하며, <A routine>은 작업을 마친 후 wakeup()를 호출하여, <B routine>이 대기 상태를 벗어나도록 해 주어야 한다.



4. Synchronization(13) - *Sleeping & Wait Queue**

(*) wait queue는 kernel mode에서 running중인 task가 특정 조건이 만족될 때까지 기다려야 할 때 사용된다.
(*) task가 필요로하는 특정 조건이나 resource가 준비될 때까지, 해당 task는 sleep 상태에 있어야 한다.

<변수 선언 및 초기화>

```
wait_queue_head_t wq;

init_waitqueue_head(&wq);
or
DECLARE_WAIT_QUEUE_HEAD(wq);
```

<Going to Sleep ➔ critical section으로 들어가기 전에 대기 상태로 빠짐>

```
wait_event                (wait_queue_head_t wq, int condition)
wait_event_interruptible  (wait_queue_head_t wq, int condition)
wait_event_killable       (wait_queue_head_t wq, int condition)
wait_event_timeout        (wait_queue_head_t wq, int condition, long timeout)
wait_event_interruptible  (wait_queue_head_t wq, int condition, long timeout)
…
```

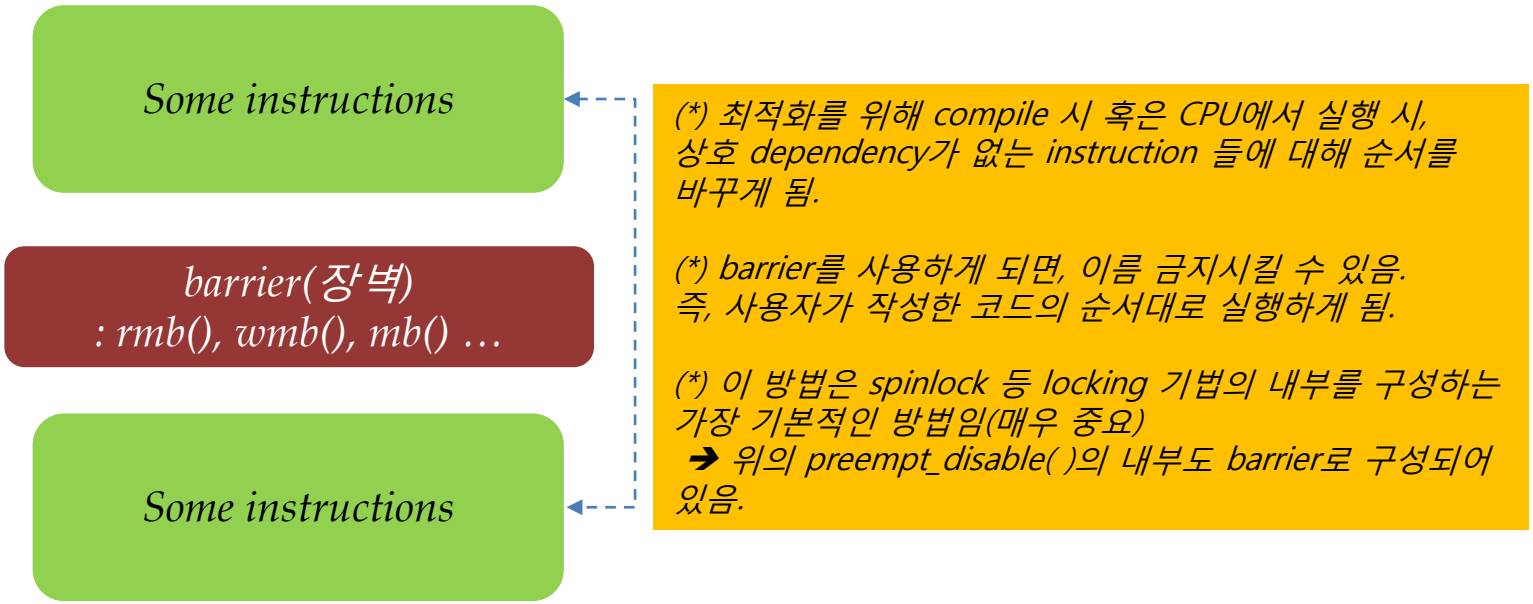
<Waking Up ➔ critical section으로 들어가는 조건을 만들어줌(풀어줌)>

```
void wake_up                      (wait_queue_head_t  *wq);
void wake_up_interruptible        (wait_queue_head_t *wq);
void wake_up_interruptible_sync   (wait_queue_head_t *wq);
…
```

4. Synchronization(14) – *Preemption Disabling & Barriers*

Preemption Disabling
(kernel preemption 금지 방법)

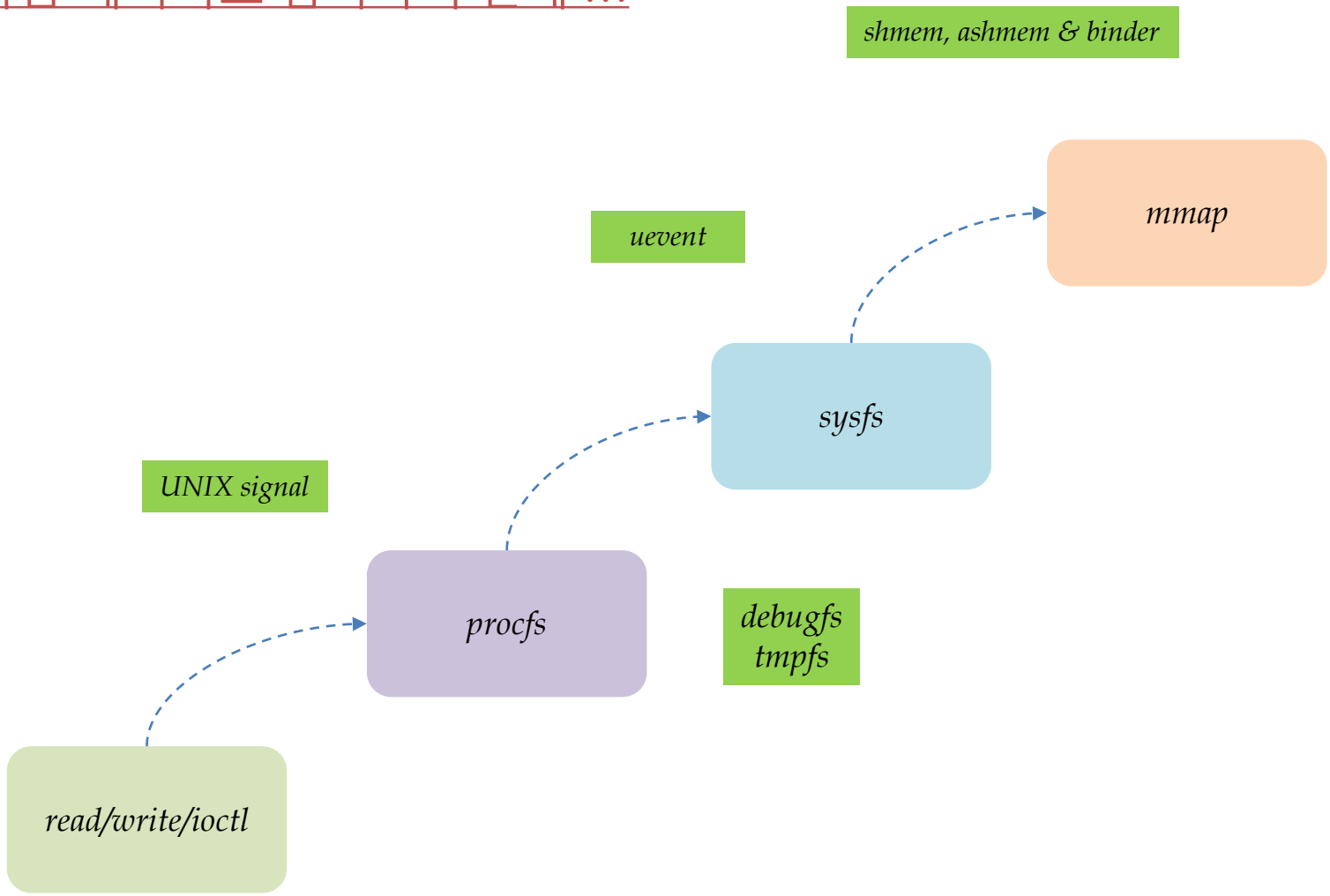
```
preempt_disable();

/* preemption is disabled .. */

preempt_enable();
```



5. Communication schemes between kernel & userspace

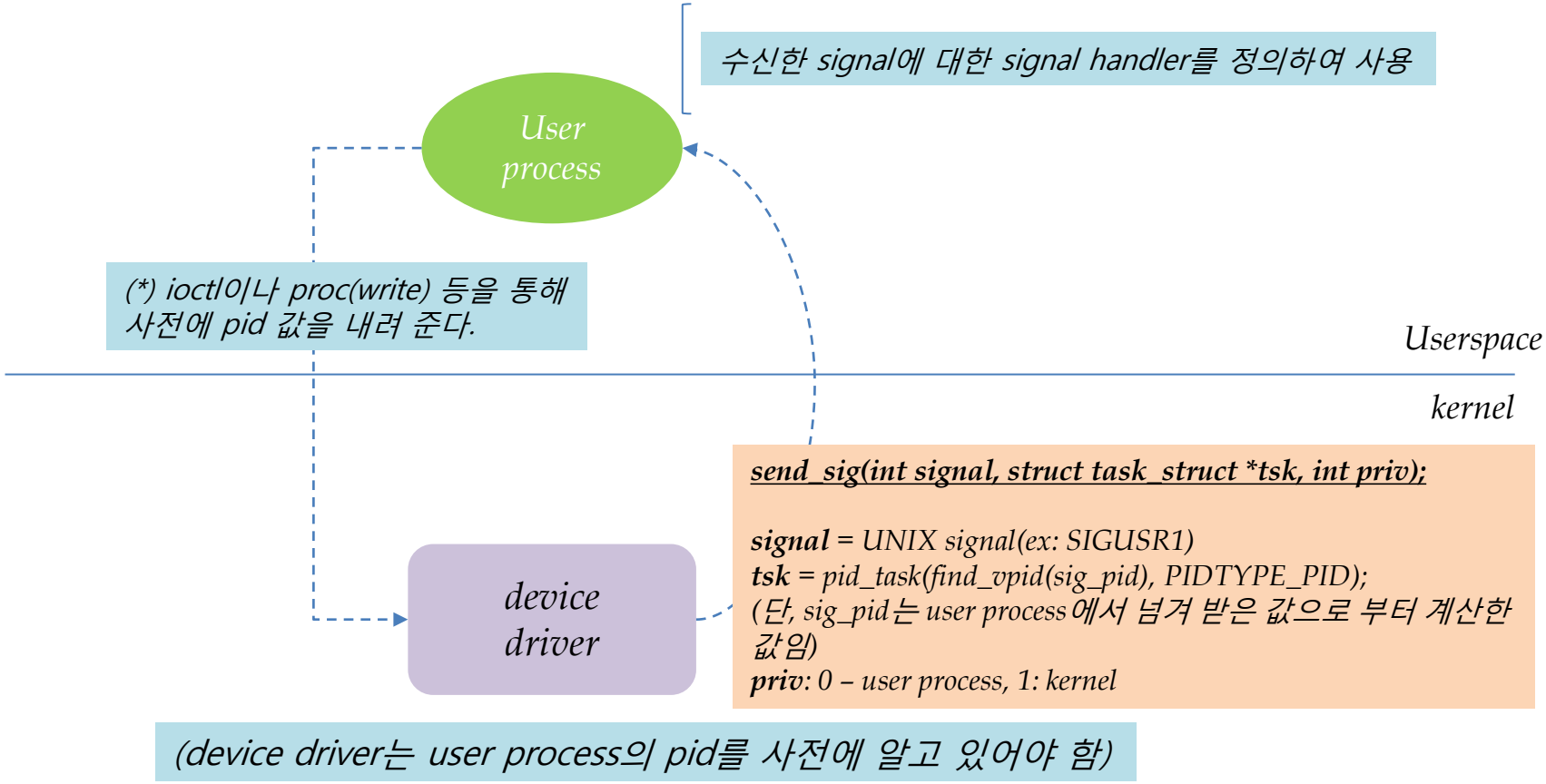
: ioctl, proc, signal, sysfs, uevent, mmap ...

(*) 다음 페이지로 넘어가지 전에 …



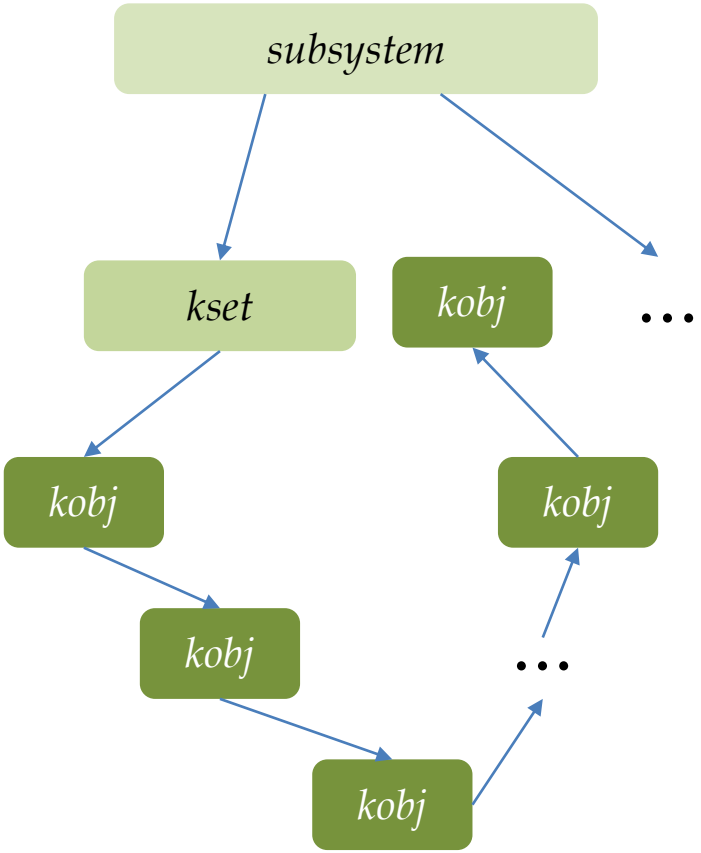
(*) 위의 화살표가 어떤 특별한 상관관계를 표현하고 있지는 않으며, 본 장에서 설명하고자 하는 전체 내용을 보여주기 위해 단순히 연결해 두었을 뿐임을 주지하기 바란다^^.
(*) 본 문서에서는 android의 중요한 주제인 binder에 관해서는 별도로 정리하지 않는다.
(*) 위의 내용 중, read/write/ioctl/procfs 등은 기본적인 사항이라 별도로 설명하지 않는다.

5.1 Send signal from kernel to userspace



(*) kernel(device driver) 내에서 특정한 사건이 발생할 경우, 이를 특정 application process에게 바로 알려 주면 매우 효과적인 경우가 있을 수 있다.
➔ 예) video decoder driver에서 buffering에 문제 발생 시, mediaserver에게 이를 알려준다...
(*) 위의 send_sig 관련 자세한 사항은 kernel/signal.c 및 include/linux/signal.h 파일 참조 !!!

5.2 kobjects & sysfs(1) - ***개념***



<관련 API 모음(arguments 생략함)>

kobject_init( )
kobject_create( )
kobject_add( )
kobject_del( )
kobject_get( )
kobject_put( )
…
sysfs_create_dir( )
sysfs_remove_dir( )
sysfs_rename_dir( )
sysfs_create_file( )
sysfs_remove_file( )
sysfs_update_file( )
sysfs_create_link( )
sysfs_remove_link( )
sysfs_create_group( )
sysfs_remove_group( )
sysfs_create_bin_file( )
sysfs_remove_bin_file( )
…

(*) kobject(kernel object)는 device model을 위해 등장한 것...
➔ Kset은 kobject의 묶음이고, subsystem은 kset의 묶음임.
(*) sysfs는 kobject의 계층 tree를 표현(view)해 주는 memory 기반의 file system으로 2.6에서 부터 소개된 방법 ➔ kernel device와 user process가 소통(통신)하는 수단. 이와 유사한 것으로 proc file system 등이 있음.
(*) kobject 관련 자세한 사항은 include/linux/kobject.h 파일 참조, sysfs 관련 자세한 사항은 include/linux/sysfs.h 파일 참조 !!!

5.2 kobjects & sysfs(2) - ***개념***

(*) sysfs는 실제로 /sys 디렉토리에 생성됨.

Internal	*External*
Kernel Objects	*Directories*
Object Attributes	*Regular Files*
Object Relationships	*Symbolic Links*

```
/sys/
|-- block
|-- bus
|-- class
|-- devices
|-- firmware
|-- module
`-- power
```

```
bus/
|-- ide
|-- pci
|-- scsi
`-- usb
```

```
class/
|-- graphics
|-- input
|-- net
|-- printer
|-- scsi_device
|-- sound
`-- tty
```

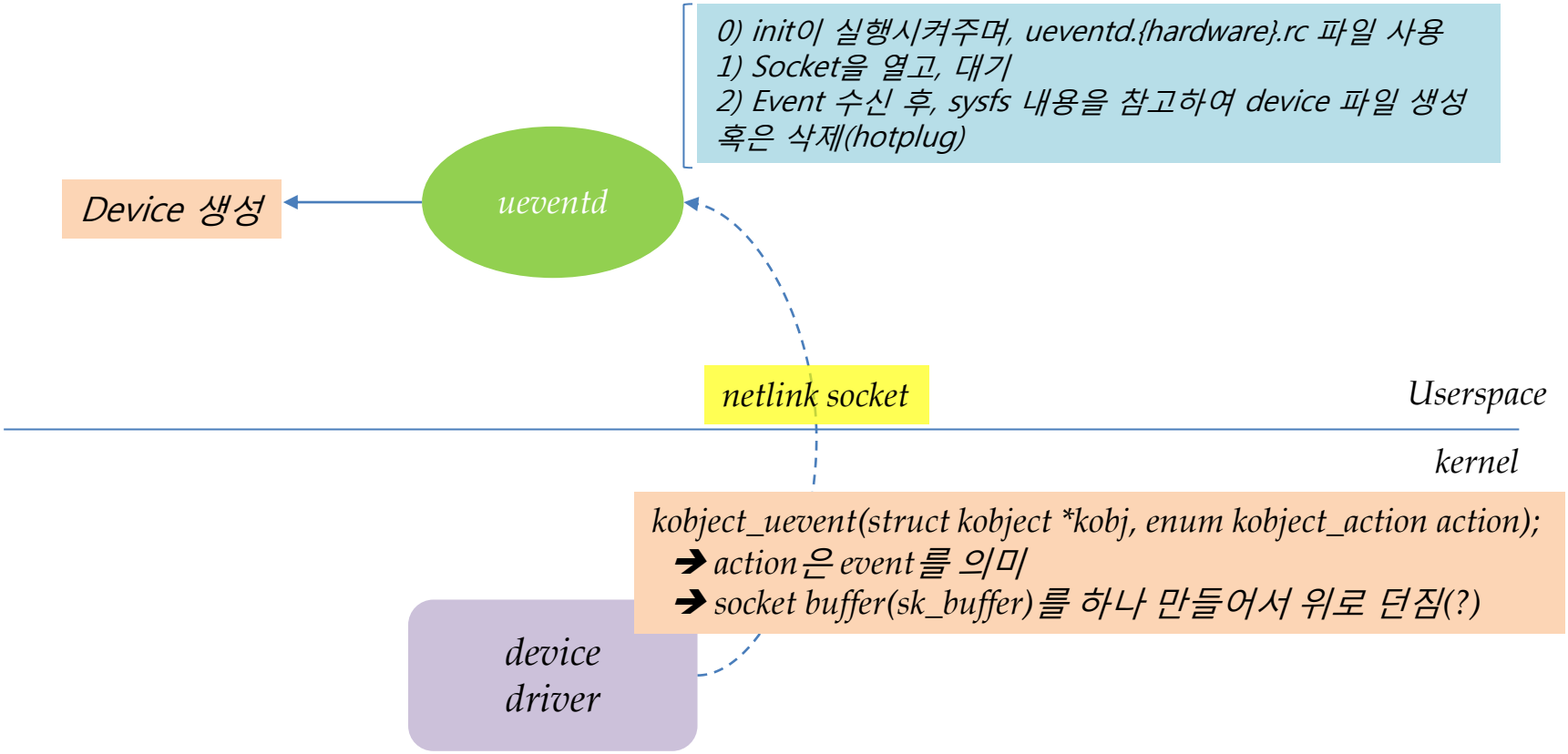
```
bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
|-- 0000:01:00.0 -> ../../../devices/pci0000:00/0000:00:01.0/0000:01:00.0
|-- 0000:02:00.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.0
|-- 0000:02:00.1 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.1
|-- 0000:02:01.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:01.0
`-- 0000:02:02.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:02.0
```

5.2 kobjects & sysfs(3) – ***간단한 사용법***

```
struct device_attribute {
  struct attribute        attr;
  ssize_t (*show)(struct device *dev, char *buf);
  ssize_t (*store)(struct device *dev, const char *buf, size_t count);
};

int device_create_file(struct device *device,
                       struct device_attribute *entry);
void device_remove_file(struct device *dev,
                        struct device_attribute *attr);
```

(*) 앞서 언급한 kobject_ 및 sysfs_ API를 이용하여 직접 작업하는 것도 가능하나, 보다 편리한 방법으로 위의 API 사용이 가능함 !
➔ 앞서 제시한 API를 사용할 경우, 매우 세세한 제어가 가능할 것임.
(*) 드라이버 초기화 시, device_create_file( )을 통해 sysfs 파일 생성이 가능하며, 드라이버 제거 시, device_remove_file( )을 통해 만들어 둔, sysfs 파일이 제거된다.
(*) device_create_file( )로 만들어둔, file을 읽고, 쓸 경우에는 각각 show 및 store에 정의한 함수가 불리어질 것이다.
(*) platform device의 경우에는, device_create_file의 첫번째 argument 값으로 .dev 필드의 정보가 전달되어야 한다.
(*) 위의 API 관련 보다 자세한 사항은 include/linux/device.h 파일 참조 !!!

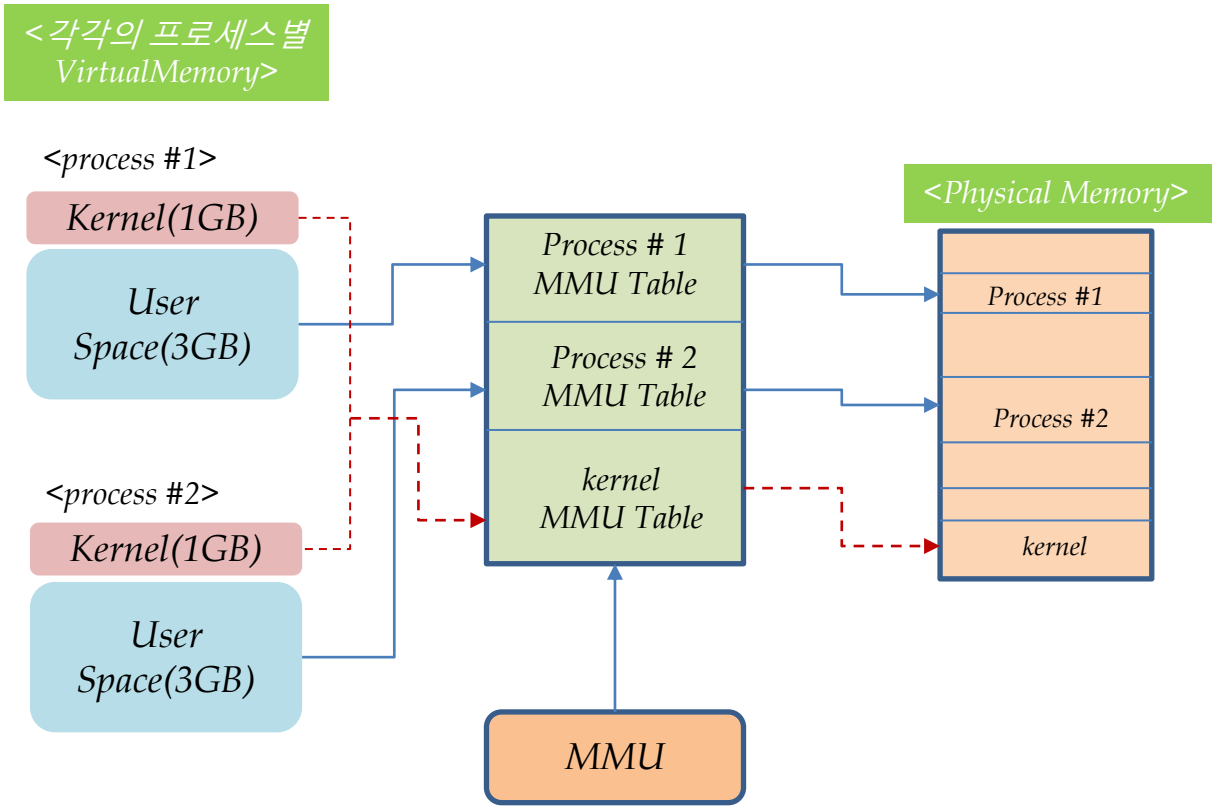
5.2 kobjects & sysfs(4) – ***uevent***



(*) 다른 통신(kernel & userspace) 방법에 비해, socket을 이용하므로 매우 편리하다.

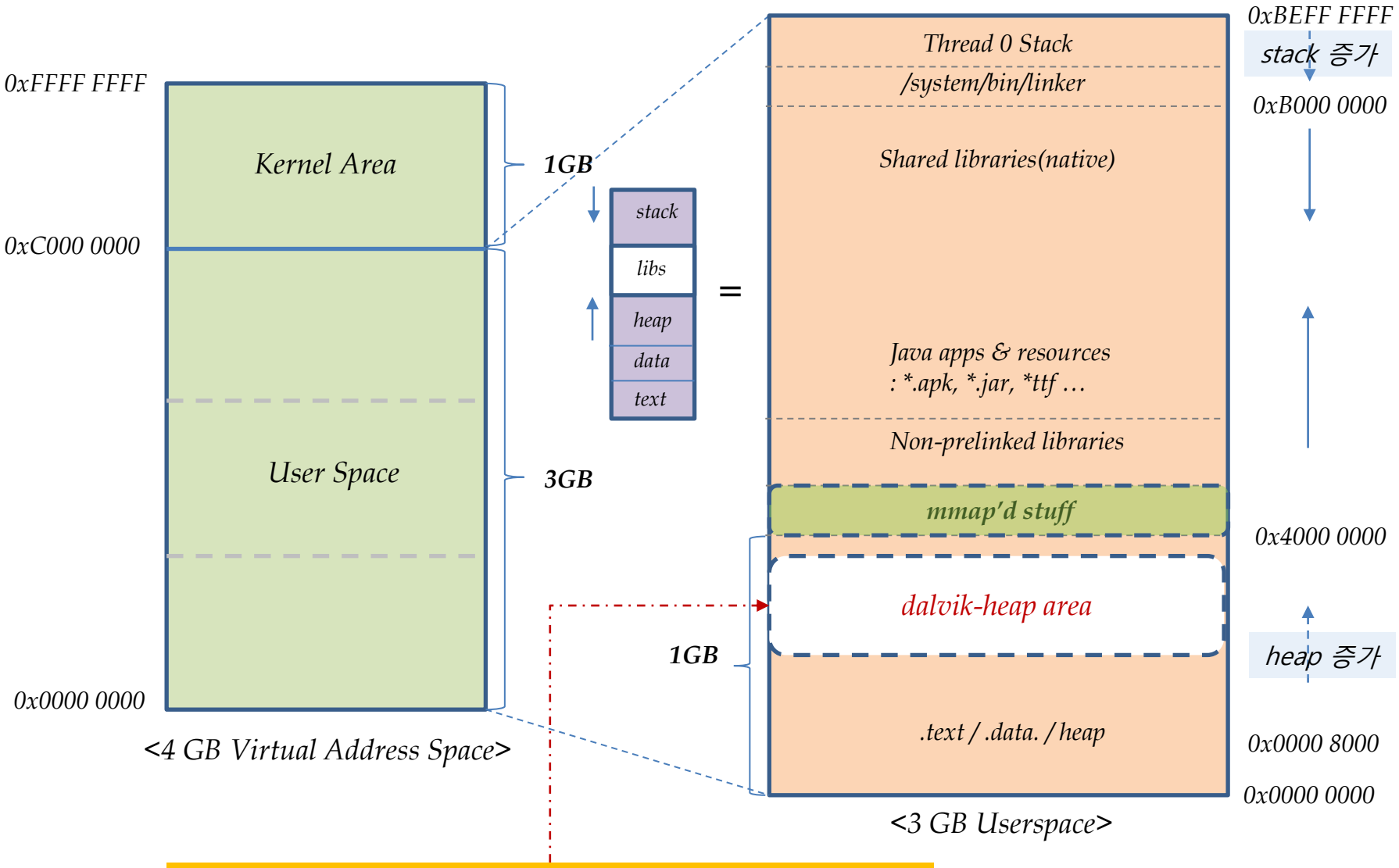
5.3 virtual memory & mmap(1) - ***background***

(*) 아래 그림은 MMU를 이용하여 Virtual Memory를 Physical Memory로 mapping하는 개념을 표현한 것임.
➔ 물리 memory가 작기 때문에 가상 memory 기법이 도입됨. 32bit CPU의 경우 2^32 = 4GB의 가상 주소 사용 가능
(*) 각각의 process는 자신만의 4GB virtual address space를 사용할 수 있다.
➔ /proc/<pid>/maps 내용을 보면, 서로 다른 process가 동일한 위치(주소)를 사용하고 있음을 알 수 있음.
(*) 아래 내용은 mmap의 원리 및 android memory map을 이해하기 위해 필요하다^^.



5.3 virtual memory & mmap(2) – *android memory map*

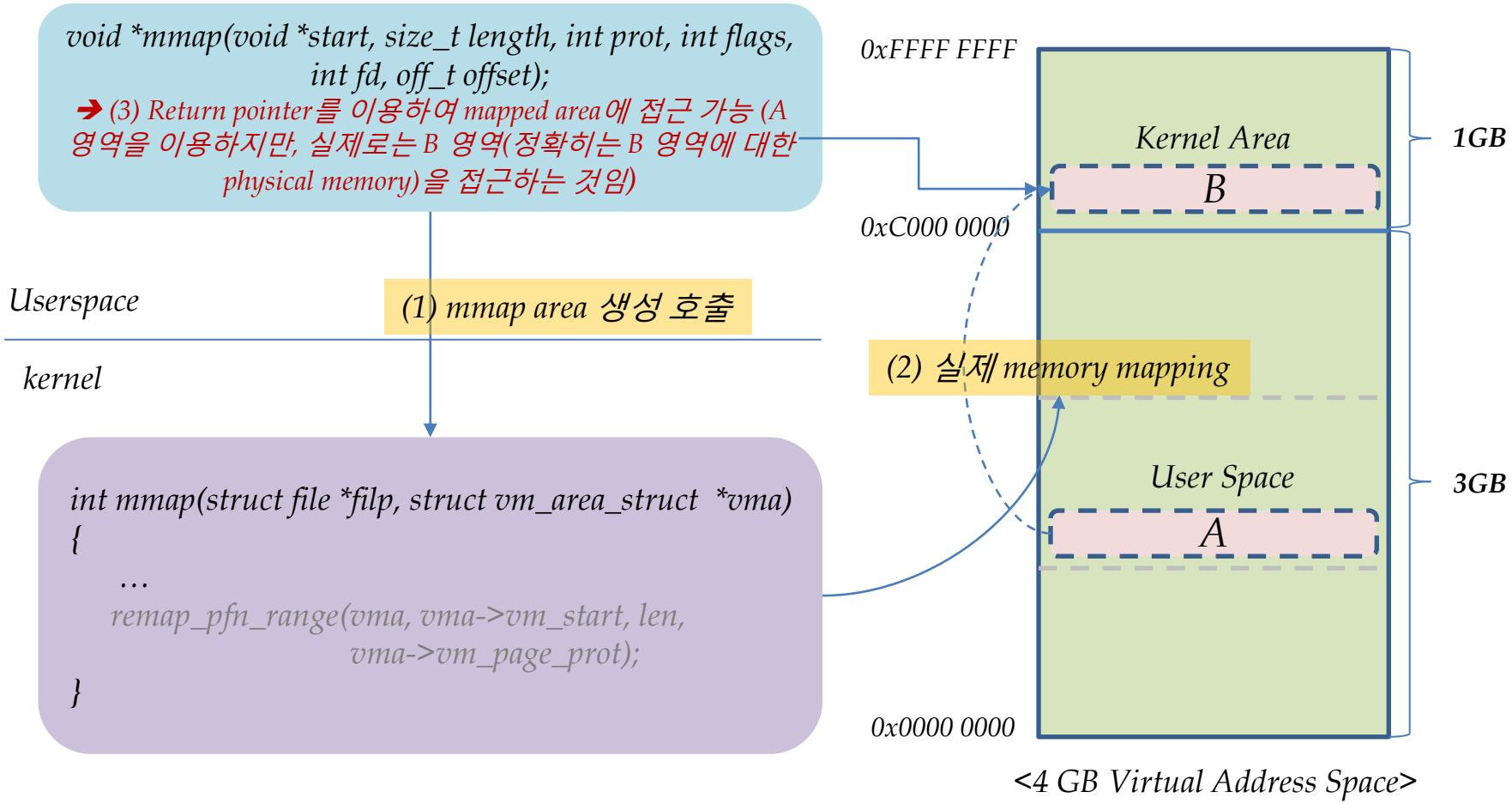
(*) 아래 User space map 정보는 prelink-linux-arm.map을 참조하여 작성한 것일 뿐, 실제 동작중인 내용(주소 값)은 다르다. (단, 각 영역의 순서/위치는 일치함)



*Dalvik VM에서 할당하는 heap의 위치는 추측일 뿐임^^*

5.3 virtual memory & mmap(3) – ***mmap***

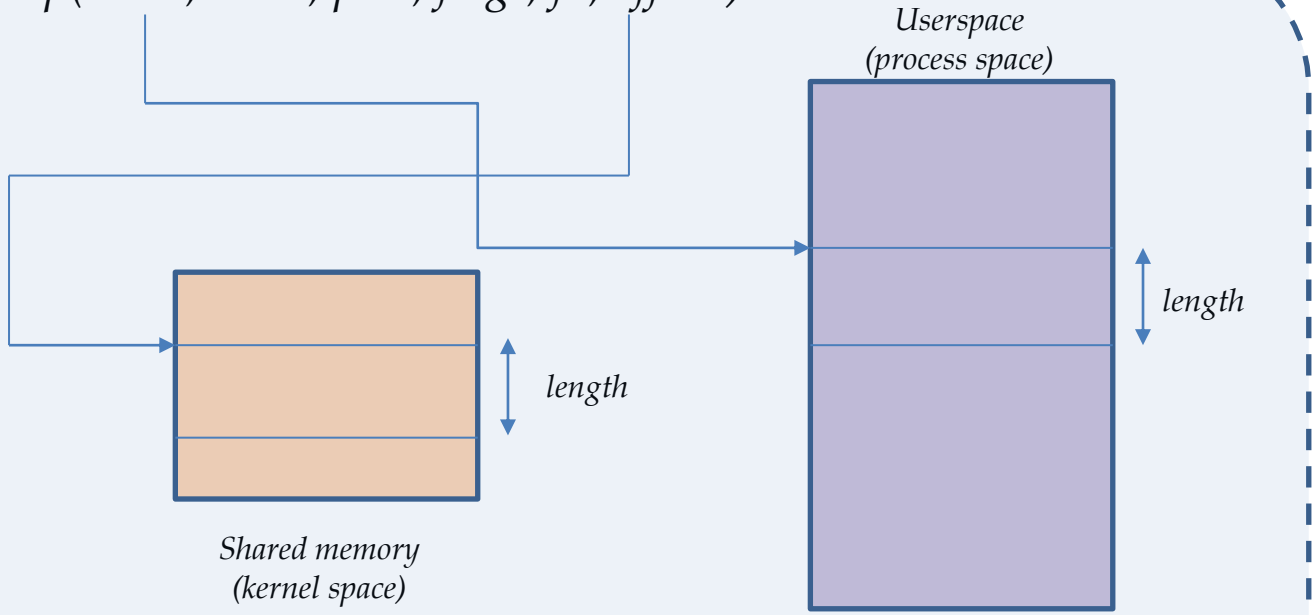
(*) mmap을 이용할 경우, application process에서 메모리 복사 과정 없이, 직접 kernel 공간을 사용할 수 있음.
➔ 반면, read/write/ioctl의 경우는 process memory와 kernel memory 사이에 메모리 copy 과정이 수반됨.
(*) device driver에서는 mmap( ) 함수 내에서 remap_pfn_range( ) 함수를 사용하여 kernel memory를 userspace 주소로 mapping 시켜 주어야 함.



5.3 virtual memory & mmap(4) - ***mmap***

(*) mmap 생성시

*void *mmap(start, lenth, prot, flags, fd, offset)*

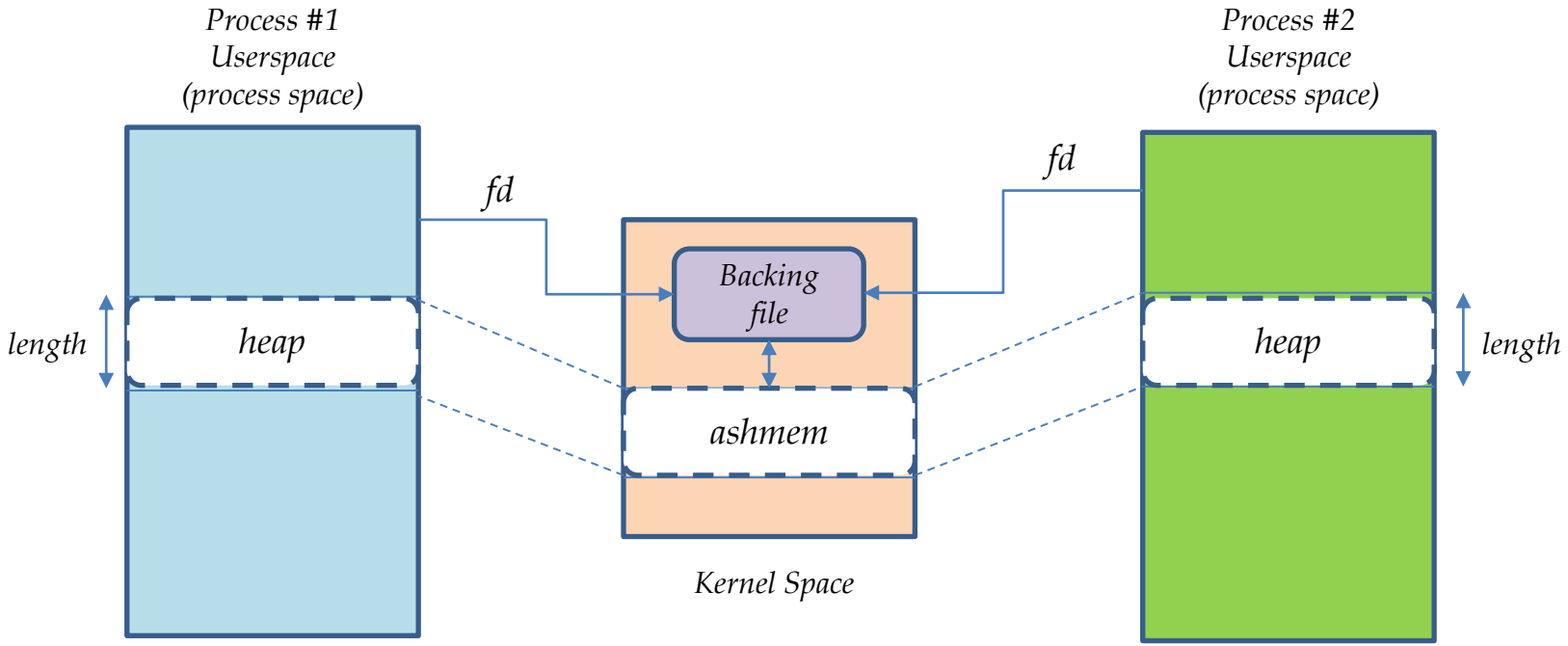


➔ *start: 일반적으로 NULL 사용*
➔ *prot: 메모리 사용 권한(PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE)*
➔ *flags: 메모리 공유 방식(MAP_FIXED, MAP_SHARED, MAP_PRIVATE)*

(*) mmap을 해제하고자 할 경우

```
int munmap (void *start, size_t length);
```

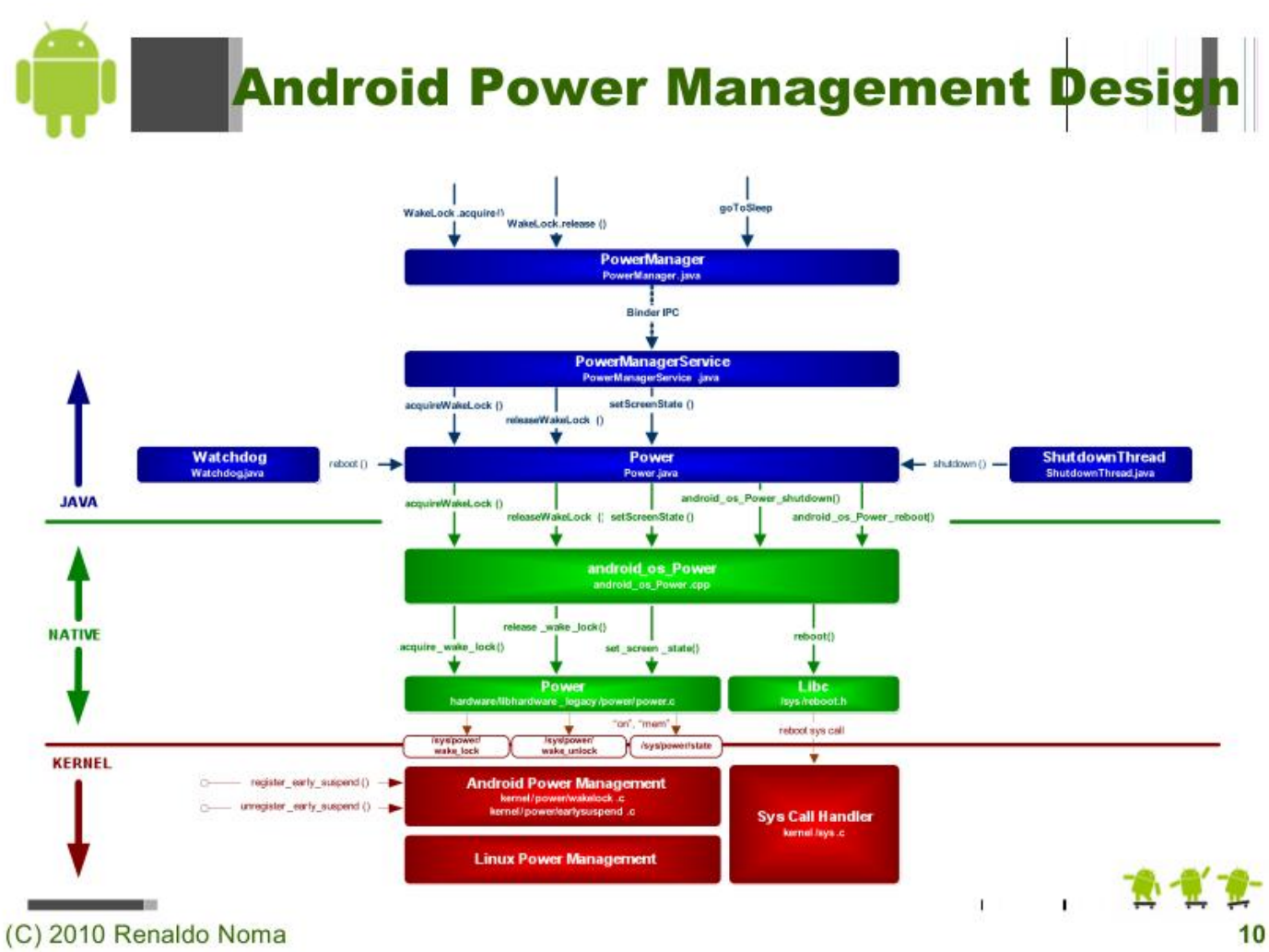
5.3 virtual memory & mmap(5) – ***ashmem***



(*) ashmem은 (named) shared memory와 유사한 기법으로, file descriptor를 통해 접근이 가능하도록 Google에서 만든 공유 메모리 기법이다.
(*) ashmem을 위해서는 내부적으로 mmap 개념이 들어가게 되며, 사용을 위해서는 binder가 필요하다.
➔ binder 관련해서는 다른 서적이나 문서를 참고하시기 바람^^.

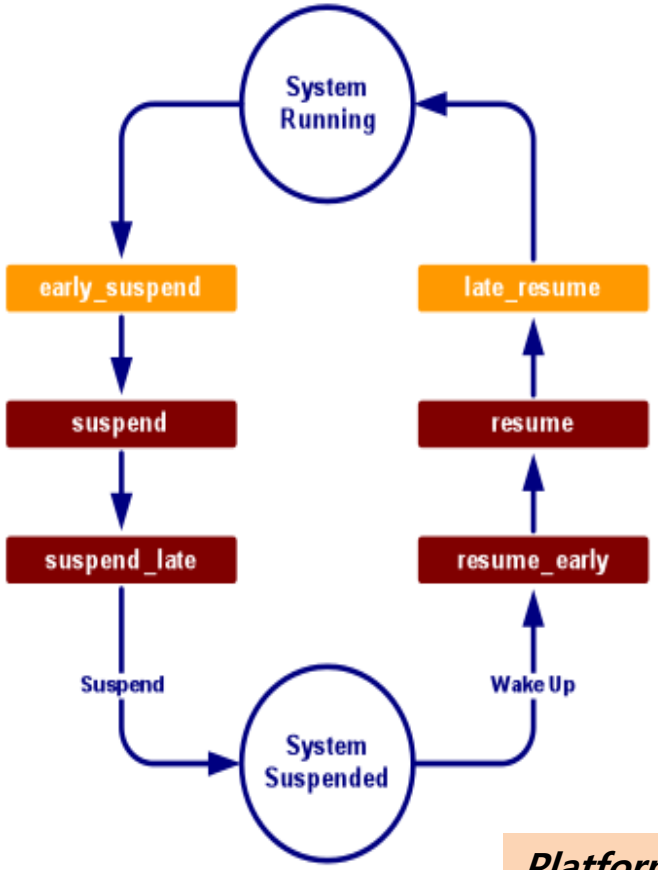
6. Suspend/Resume and Wakelock(1)

(*) 이 그림은 Android 전체 power manangement를 이해하기 위해 첨부하였다.



(*) /sys/power/state 파일을 통해 kernel의 전원 서비스를 이용 가능함.

6. Suspend/Resume and Wakelock(2) – *suspend/resume(1)*




```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*suspend_late)(struct platform_device *, pm_message_t state);
        int (*resume_early)(struct platform_device *);
        int (*resume)(struct platform_device *);
};
```

```
struct device_driver {
        int (*probe) (struct device *dev);
        int (*remove) (struct device *dev);
        void (*shutdown) (struct device *dev);
        int (*suspend) (struct device *dev, pm_message_t state);
        int (*resume) (struct device *dev);
};
```

- Any driver can register its own early_suspend and late_resume handler using register_early_suspend() API
- Unregistration is done using unregister_early_suspend() API

Platform hooks

arch/arm/mach-xxx/pm.c 파일내의 platform_suspend_ops 함수를 이용하여 등록함.

➔ ***prepare(), enter(), finish(), valid()***

➔ ***Enter_state 함수에 의해 suspend_ops 함수가 호출될 것임.***

➔ ***Enter_state 함수를 구동 시키기 위해서는***
echo mem > /sys/power/state

6. Suspend/Resume and Wakelock(3) – ***suspend/resume(2)***

Suspend:

1) 프로세스와 task를 freezing 시키고,
2) 모든 device driver의 suspend callback 함수 호출
3) CPU와 core device를 suspend 시킴

Resume:

1) System 장치(/sys/devices/system)를 먼저 깨우고,
2) IRQ 활성화, CPU 활성화
3) 나머지 모든 장치를 깨우고, freezing되어 있는 프로세스와 task를 깨움.

Early Suspend: google에서 linux kernel에 추가한 새로운 상태로, linux의 original suspend 상태와 LCD screen off 사이에 존재하는 새로운 상태를 말한다. LCD를 끄면 배터리 수명과 몇몇 기능적인 요구 사항에 의해 LCD back-light나, G-sensor, touch screen 등이 멈추게 된다.

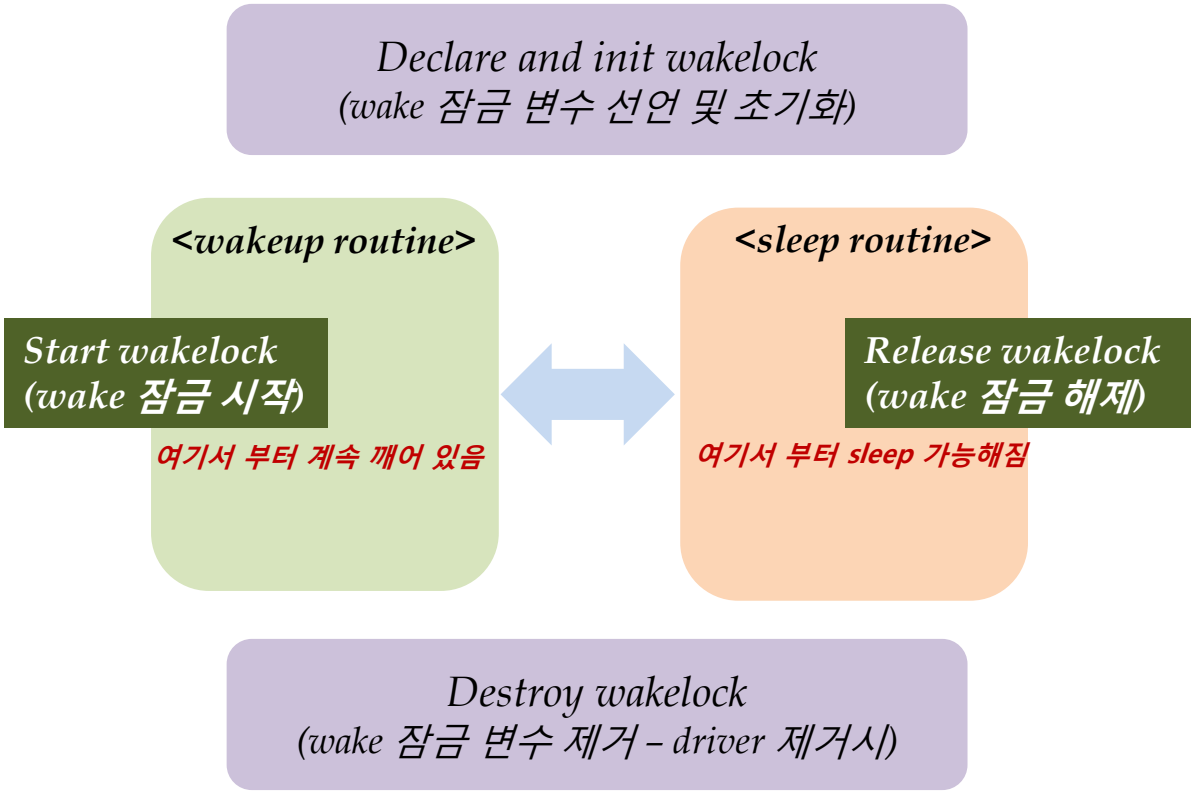
Late Resume: Early Suspend와 쌍을 이루는 새로운 상태로, 역시 google에서 linux kernel에 추가하였다. Linux resume이 끝난 후 수행되며, early suspend 시 꺼진 장치들이 resume하게 된다.

(*) suspend/resume 및 early suspend/late resume 관련 내용은 아래 파일에서 확인할 수 있다.

1) kernel/power/main.c
2) kernel/power/earlysuspend.c
3) kernel/power/wakelock.c
4) arch/arm/mach-xxx/pm..c

6. Suspend/Resume and Wakelock(4) - ***wakelock***

*(*) **wakelock**: android 전원 관리 시스템의 핵심을 이루는 기능으로, 시스템이 low power state로 가는 것을 막아주는 메카니즘(google에서 만듦)이다.*
(*) Smart Phone은 전류를 많이 소모하므로, 항시 sleep mode로 빠질 준비를 해야 한다.
(*) wake_lock_init의 인자로 넘겨준, name 값은 /proc/wakelocks에서 확인 가능함.



6. Suspend/Resume and Wakelock(5) - *wakelock*

<Wakelock 관련 API 모음>

[변수 선언] struct wakelock mywakelock;

[초기화] **wake_lock_init**(&mywakelock, int type, "wakelock_name");

➔ type :

= WAKE_LOCK_SUSPEND: 시스템이 suspending 상태로 가는 것을 막음

= WAKE_LOCK_IDLE: 시스템이 low-power idle 상태로 가는 것을 막음.

[To hold(wake 상태로 유지)] **wake_lock**(&mywakelock);

[To release(sleep 상태로 이동)] **wake_unlock**(&mywakelock);

[To release(sleep 상태로 이동)] **wake_lock_timeout**(&mywakelock, HZ);

[제거] **wake_lock_destroy** (&mywakelock);

6. Suspend/Resume and Wakelock(6) – ***runtime power management***

<TODO>

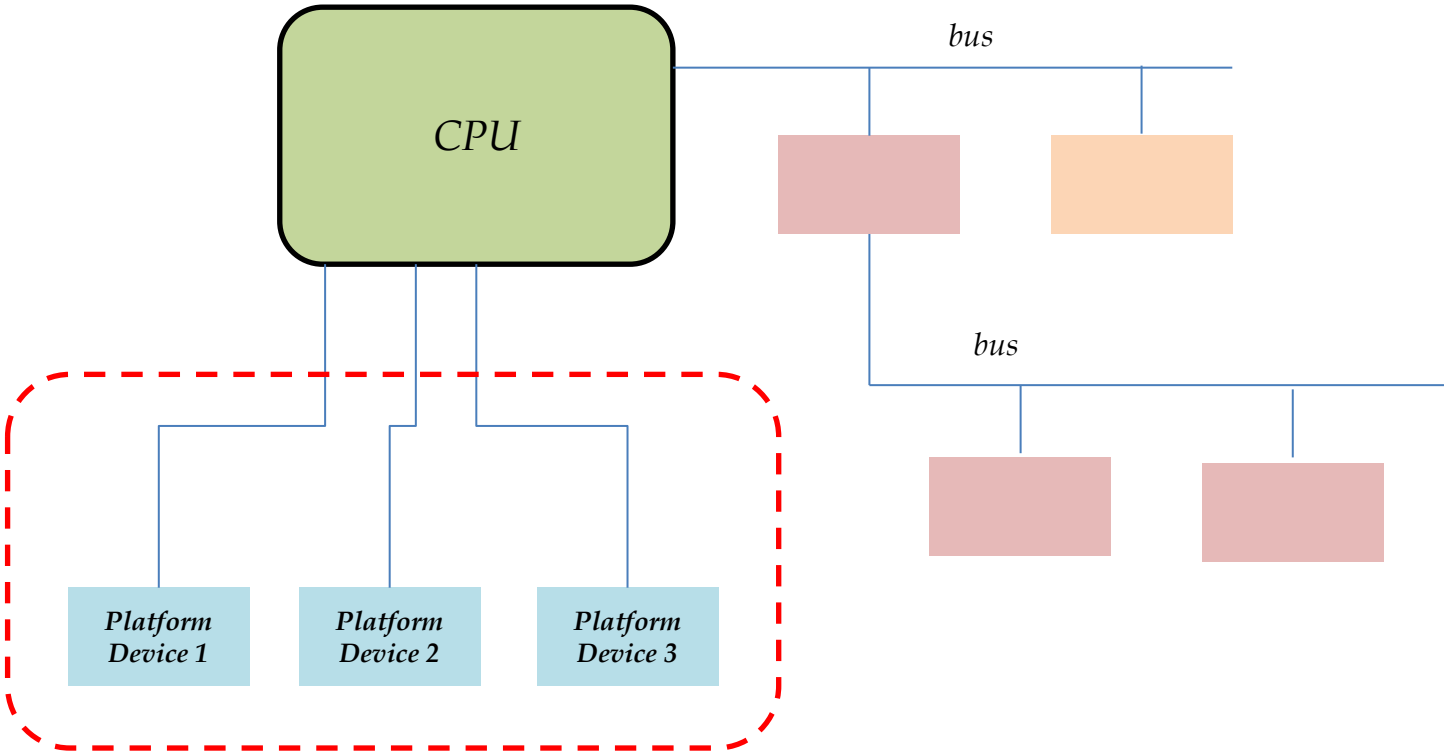
(*) I/O 장치들에 대해, run-time에 low-power state로 만들거나, wake-up 시키는 것을 run-time power management라고 함,
(*) PM core에서 아래의 callback 함수를 정의(등록)한 드라이버에 대해 작업 수행한다.
(*) include/linux/pm.h 파일 참조

```
struct dev_pm_ops {
   …
   int (*runtime_suspend)(struct device *dev);
   int (*runtime_resume)(struct device *dev);
   int (*runtime_idle)(struct device *dev);
   …
};
```

(*) 보다 자세한 사항은 Documentation/power/runtime_pm.tx 파일 참조 !

7. Platform Device & Driver(1) - ***개념***

- 1) Embedded system의 시스템의 경우, bus를 통해 device를 연결하지 않는 경우가 있음.
  ➔ *bus는 확장성(enumeration), hot-plugging, unique identifier를 허용함에도 불구하고 …*
- 2) platform driver/platform device infrastructure를 사용하여 이를 해결할 수 있음.
- ➔ *platform device란, 별도의 bus를 거치지 않고, CPU에 직접 연결되는 장치를 일컬음.*



7. Platform Device & Driver(2) - 개념

- platform_device 정의 및 초기화
- resource 정의

(arch/arm/mach-msm/board-XXXX.c 파일에 위치함)

<예 – bluetooth sleep device>

```
struct platform_device my_bluesleep_device = {
    .name  = "bluesleep",
    .id    = 0,
    .num_resources = ARRAY_SIZE(bluesleep_resources),
    .resource = bluesleep_resources,
};
```

.name 필드("bluesleep")로 상호 연결

- platform_driver 정의 및 초기화
- probe/remove

(drivers/XXXX/xxxx.c 등에 위치함)

```
struct platform_driver bluesleep_driver = {
    .remove = bluesleep_remove,
    .driver = {
        .name = "bluesleep",
        .owner = THIS_MODULE,
    },
};
```

(*) drivers/base/platform.c
(*) include/linux/platform_device.h 참조

7. Platform Device & Driver(3) – *platform driver*

- *(*) drivers/serial/imx.c file에 있는 iMX serial port driver를 예로써 소개하고자 함. 이 드라이버는 platform_driver structure를 초기화함.*

```
static struct platform_driver serial_imx_driver = {
        .probe          = serial_imx_probe,
        .remove         = serial_imx_remove,
        .driver         = {
                .name   = "imx-uart",
                .owner  = THIS_MODULE,
        },
};
```

- *(*) init/cleanup시, register/unregister 하기*

```
static int __init imx_serial_init(void)
{
        platform_driver_register(&serial_imx_driver);
}
static void __ext imx_serial_cleanup(void)
{
        platform_driver_unregister(&serial_imx_driver);
}
```

7. Platform Device & Driver(4) – *platform_device*

- *(*) 플랫폼 디바이스는 동적으로 감지(detection)가 될 수 없으므로, static하게 지정해 주어야 함. static하게 지정하는 방식은 chip 마다 다를 수 있는데, ARM의 경우는 board specific code (arch/arm/mach-imx/mx1ads.c)에서 객체화 및 초기화(instantiation)를 진행하게 됨.*

- *(*) Platform 디바이스와 Platform 드라이버를 matching시키기 위해서는 name(아래의 경우는 "imx-uart")을 이용함.*

```
static struct platform_device imx_uart1_device = {
        .name           = "imx-uart",
        .id             = 0,
        .num_resources  = ARRAY_SIZE(imx_uart1_resources),
        .resource       = imx_uart1_resources,
        .dev = {
                .platform_data = &uart_pdata,
        }
};
```

7. Platform Device & Driver(5) - *platform_device(초기화)*

- *(*) platform device는 아래 list에 추가되어야 함.*

```
static struct platform_device *devices[] __initdata = {
        &cs89x0_device,
        &imx_uart1_device,
        &imx_uart2_device,
};
```

- *(*) platform_add_devices() 함수를 통해서 실제로 시스템에 추가됨.*

```
static void __init mx1ads_init(void)
{
        [...]
        platform_add_devices(devices, ARRAY_SIZE(devices));
        [...]
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
        [...]
        .init_machine   = mx1ads_init,
MACHINE_END
```

7. Platform Device & Driver(6) - *platform_device(resource)*

- *(*) 특정 드라이버가 관리하는 각 장치(device)는 서로 다른 H/W 리소스를 사용하게 됨.*
  ➔ I/O 레지스터 주소, DMA 채널, IRQ line 등이 서로 상이함.

  (*) 이러한 정보는 struct resource data structure를 사용하여 표현되며, 이들 resource 배열은 platform device 정의 부분과 결합되어 있음.
- *(*) platform driver내에서 platform_device 정보(pointer)를 이용하여 resource를 얻어오기 위해서는 platform_get_resource_byname(...) 함수가 사용될 수 있음.*

```
static struct resource imx_uart1_resources[] = {
        [0] = {
                .start  = 0x00206000,
                .end    = 0x002060FF,
                .flags  = IORESOURCE_MEM,
        },
        [1] = {
                .start  = (UART1_MINT_RX),
                .end    = (UART1_MINT_RX),
                .flags  = IORESOURCE_IRQ,
        },
};
```

7. Platform Device & Driver(7) - *platform_device(device specific data)*

- *(*) 앞서 설명한 resource data structure 외에도, 드라이버에 따라서는 자신만의 환경 혹은 데이터(configuration)을 원할 수 있음.이는 struct platform_device 내의 platform_data를 사용하여 지정 가능함.*
  *(*) platfor_data는 void * pointer로 되어 있으므로, 드라이버에 임의의 형식의 데이타 전달이 가능함.*
  (*) iMX 드라이버의 경우는 struct imxuart platform data가 platform_data로 사용되고 있음.

```
static struct imxuart_platform_data uart_pdata = {
        .flags = IMXUART_HAVE_RTSCTS,
};
```

7. Platform Device & Driver(8) – *platform driver(probe, remove)*

- *(*) 보통의 probe함수 처럼, 인자로 platform_device에의 pointer를 넘겨 받으며, 관련 resource를 찾기 위해 다른 utility 함수를 사용하고, 상위 layer로 해당 디바이스를 등록함. 한편 별도의 그림으로 표현하지는 않았으나, probe의 반대 개념으로 드라이버 제거 시에는 remove 함수가 사용됨.*

```
static int serial_imx_probe(struct platform_device *pdev)
{
        struct imx_port *sport;
        struct imxuart_platform_data *pdata;
        void __iomem *base;
        struct resource *res;

        sport = kzalloc(sizeof(*sport), GFP_KERNEL);
        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        base = ioremap(res->start, PAGE_SIZE);

        sport->port.dev = &pdev->dev;
        sport->port.mapbase = res->start;
        sport->port.membase = base;
        sport->port.type = PORT_IMX,
        sport->port.iotype = UPIO_MEM;
        sport->port.irq = platform_get_irq(pdev, 0);
        sport->rxirq = platform_get_irq(pdev, 0);
        sport->txirq = platform_get_irq(pdev, 1);
        sport->rtsirq = platform_get_irq(pdev, 2);

        [...]
```

References

- *1) Linux Kernel Development(3rd edition) ……..… [Robert Love]*
- *2) Writing Linux Device Drivers …… [Jerry Cooperstein]*
- *3) Essential Linux Device Drivers ……..… [Sreekrishnan Venkateswaran]*
- *4) Linux kernel 2.6 구조와 원리 ……... [이영희 역, 한빛미디어]*
- *5) Linux Kernel architecture for device drivers ……..*
  [Thomas Petazzoni Free Electronics(thomas.petazzoni@free-electronics.com)]
- *6) The sysfs Filesystem ……..……. [Patrick Mochel, mochel@digitalimplant.org]*
- *7) Linux SD/MMC Driver Stack …..... [Champ Yen, champ.yen@gmail.com]*
- *8) %233.GTUG-Android-Power Management.pdf … [Renaldo Noma 2010]*
- *9) Android_Debug_Guide6.pdf ……. [Chunghan Yi]*
- *10) 안드로이드 아나토미 시스템 서비스 ……. [김태연/박지훈/김상엽/이왕재, 개발자가 행복한 세상]*
- *11) InterruptThreads-Slides_Anderson.ppt …… [Mike Anderson, mike@theptrgroup.com]*
- *12) Some Internet Articles …*

Thanks a lot !

SlowBoot