

*Android(Linux) **Kernel** Hacks2*

: SMP Boot, ARMIRQ, CPU Hotplug, WorkQueue, AutoSleep/Wakeup Source, ...



ANDROID

chunghan.yi@gmail.com, slowboot

본 문서에서는 ...

- 1) Android mobile phone의 전반적인 boot flow를 한눈에 확인할 수 있는가 ?
- 2) Chip 제조사에서 제공하는 SoC를 초기화 부분(도입부)이 잘 기술되어 있는가 ?
- 3) SMP 관련(smp boot, hotplug, interrupt 등)하여 적절히 설명되고 있는가 ?
- 4) ARM IRQ의 구조를 잘 표현하고 있는가 ?
- 5) CPU Hotplug의 개념이 잘 표현되어 있는가 ?
- 6) 최신 work queue의 구조가 잘 설명되어 있는가 ?
- 7) Driver 초기화 부분이 잘 기술되어 있는가 ?
- 8) Memory management 관련 핵심 point가 잘 기술되어 있는가 ?
- 9) 최신 Power Management 기법(auto sleep, wakeup sources)이 잘 설명되어 있는가 ?

- 1) kernel 3.8.x를 참조하였다(단, 편의상 일부 내용 중에는 예전 version 내용이 포함되어 있음).
- 2) ARM architecture(특히 Qualcomm chip)를 기반으로 설명하고자 하였다.
 - ➔ 단, 기업 비밀에 해당하는 자세한 사항은 당연히 언급하지 않았으며, 논리 전개상 필요한 부분에 한하여 rough하게 정리하였음.
- 3) Android는 Jelly Bean 4.x를 대상으로 하였다.

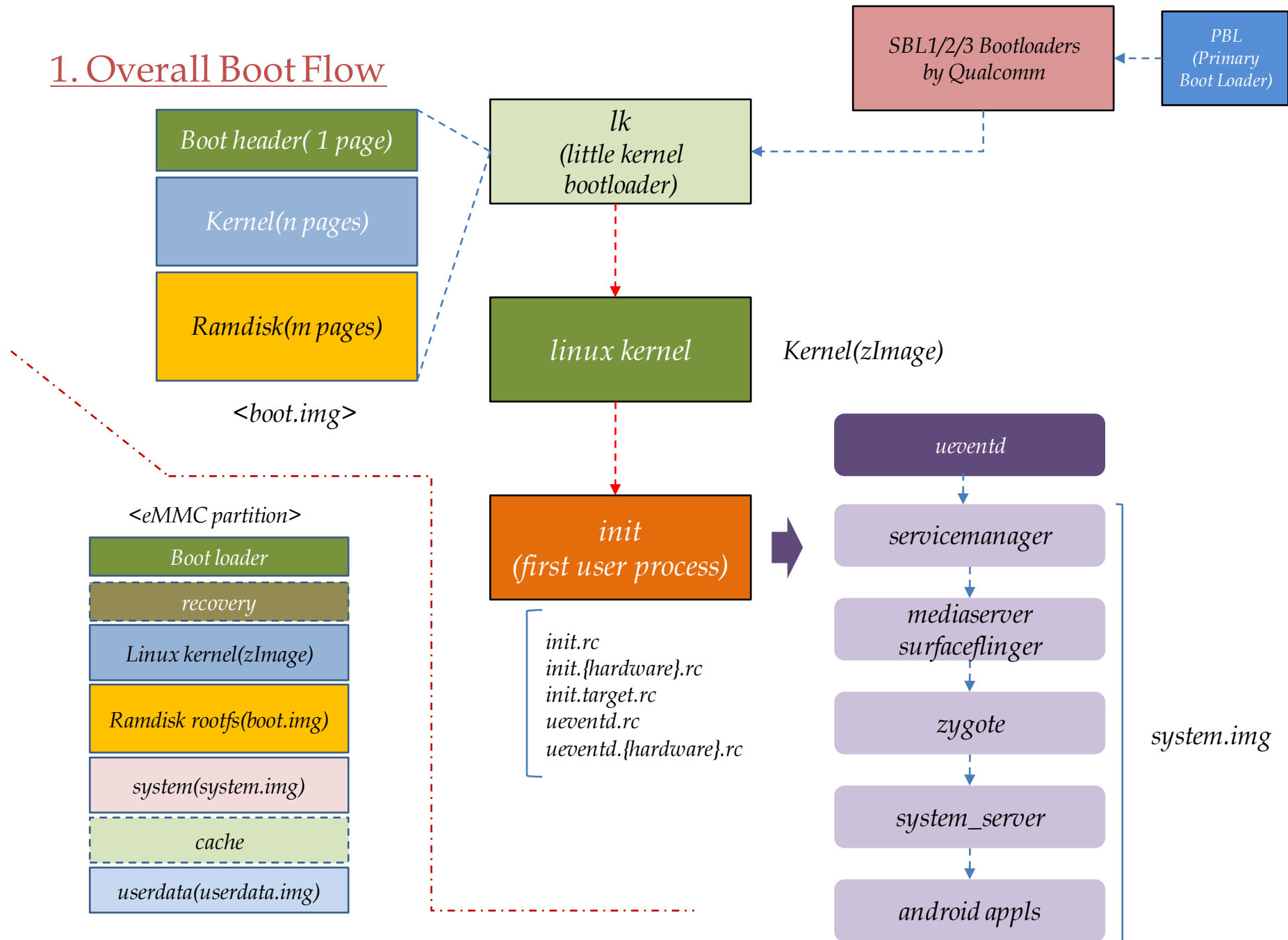
(*) 본 문서는 Android_Device_Driver_GuideX.pdf에 빠진 내용을 정리할 목적으로 작성되었으며, 내용 중 일부는 사실과 다를 수 있음 ^^.

목차

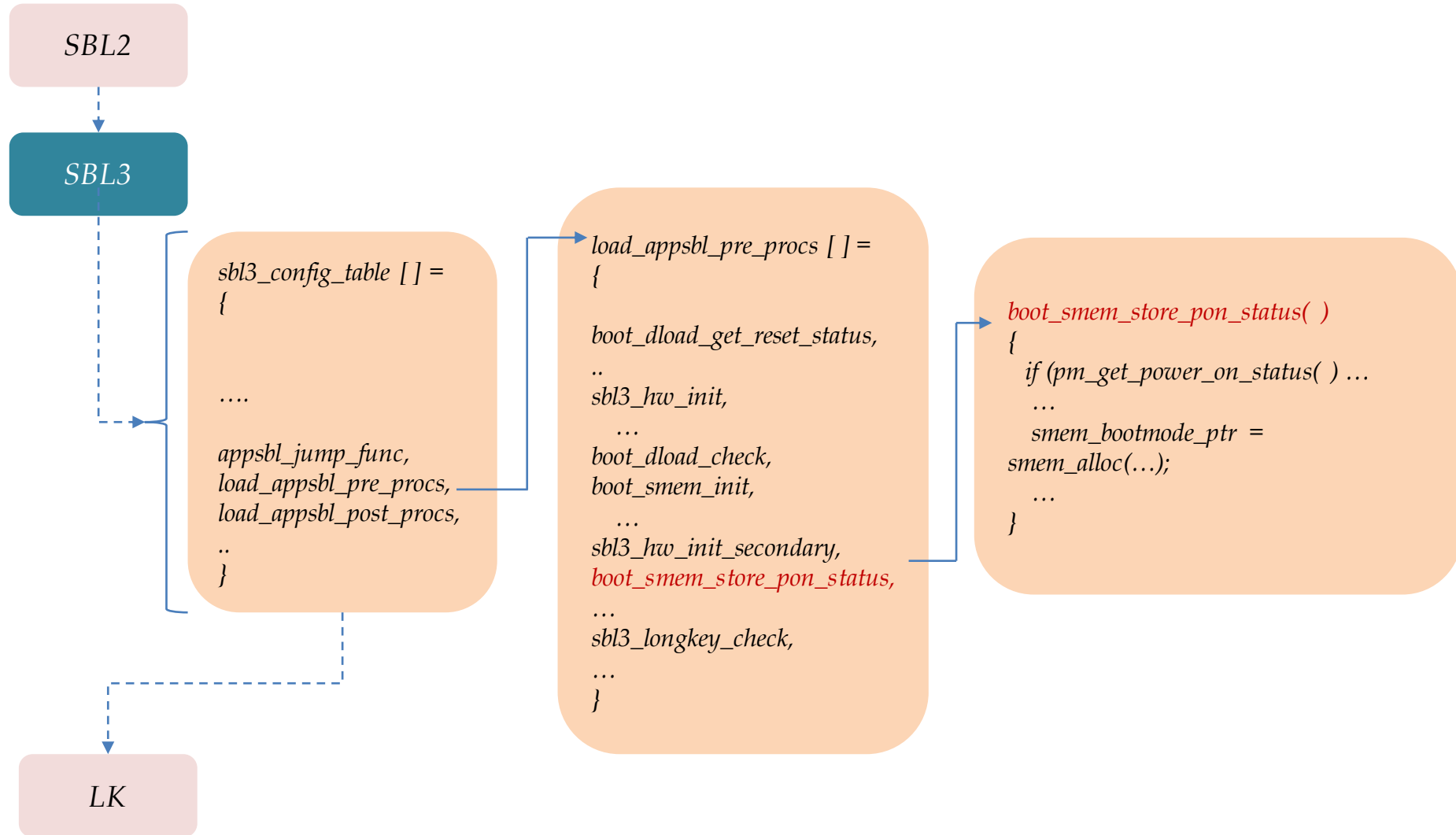
- 1. Boot Boot Boot
- 2. Machine Description
- 3. ARM IRQ
- 4. CPU Hotplug
- 5. Work Queue
- 6. Driver Initialization
- 7. Power Management
- 8. Memory Management
- 9. Notifier
- **References**

Boot Boot Boot

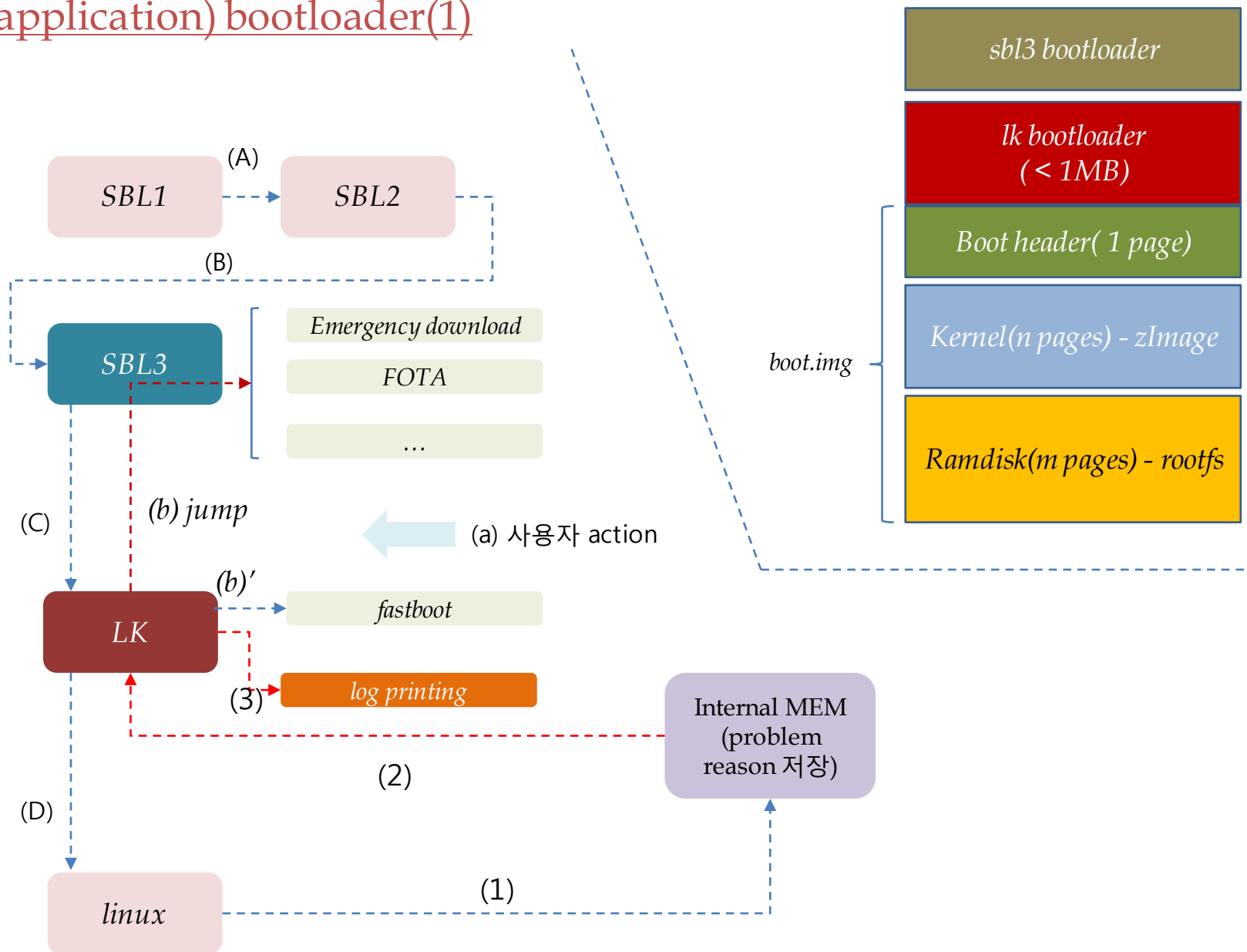
1. Overall Boot Flow



2. Secondary Bootloader – *SBL3 Overview*



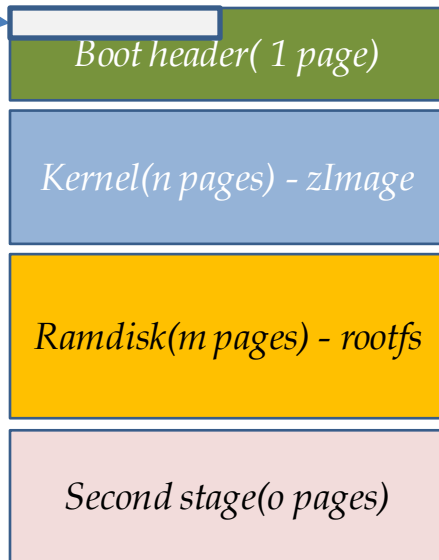
3. lk(application) bootloader(1)



3. lk(application) bootloader(2)

lk/app/aboot/bootimg.h

ANDROID!



lk/app/aboot/aboot.c

`partition_get_index("boot")`

`mmc_read(raw_header, 2048)`

`Check BOOT_MAGIC`

`mmc_read(hdr->kernel_addr)`

`mmc_read(hdr->ramdisk_addr)`

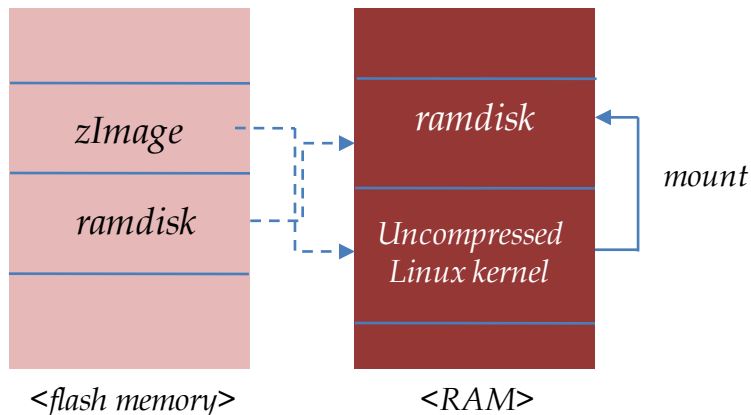
`boot_linux_from_mmc()`

`create_atags(cmdline, ramdisk)`

`entry(0, board_type,
ADDR_TAGS
<= kernel_addr`

`boot_linux()`

Linux kernel starts...

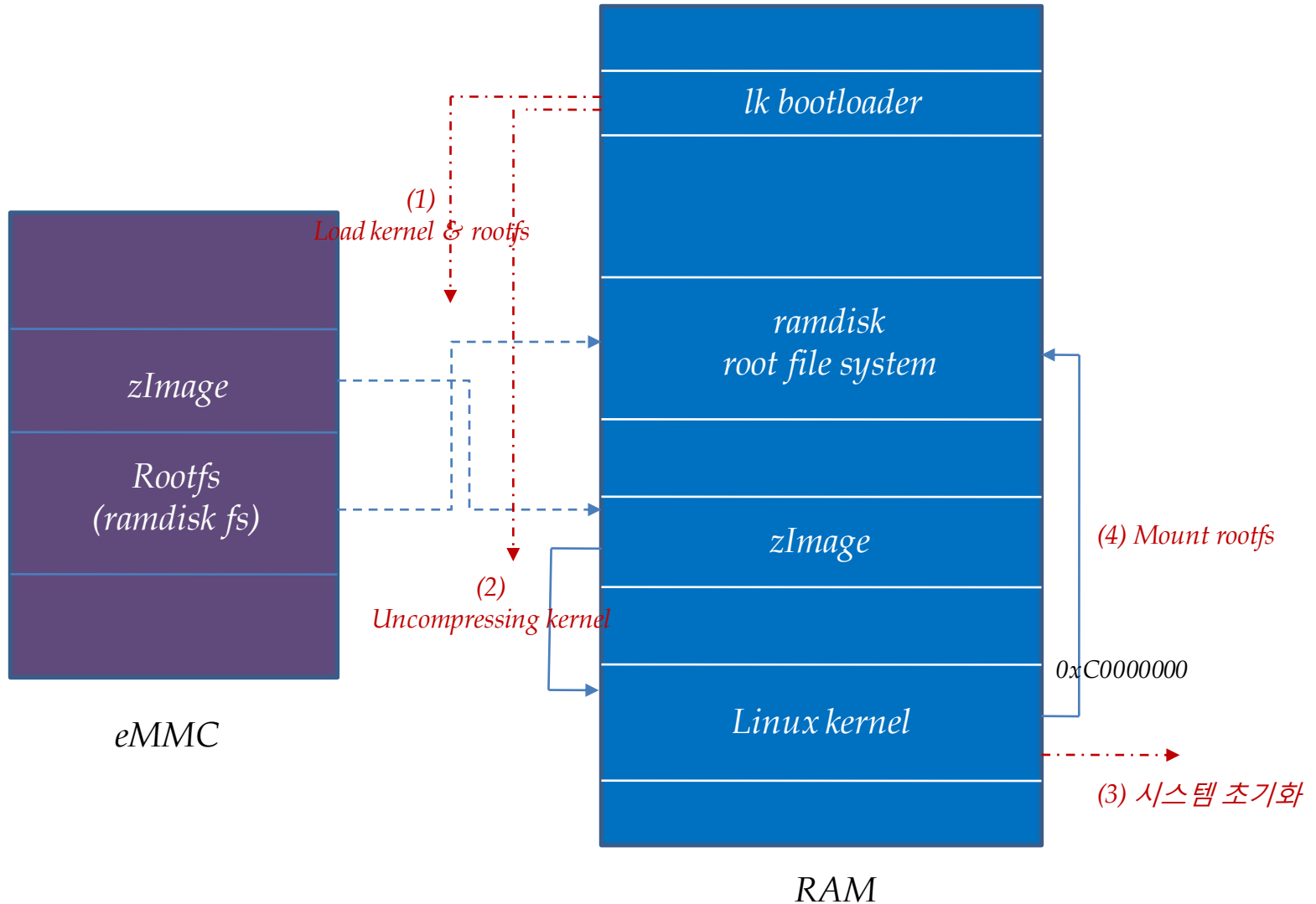


4. Linux Kernel Boot(1)

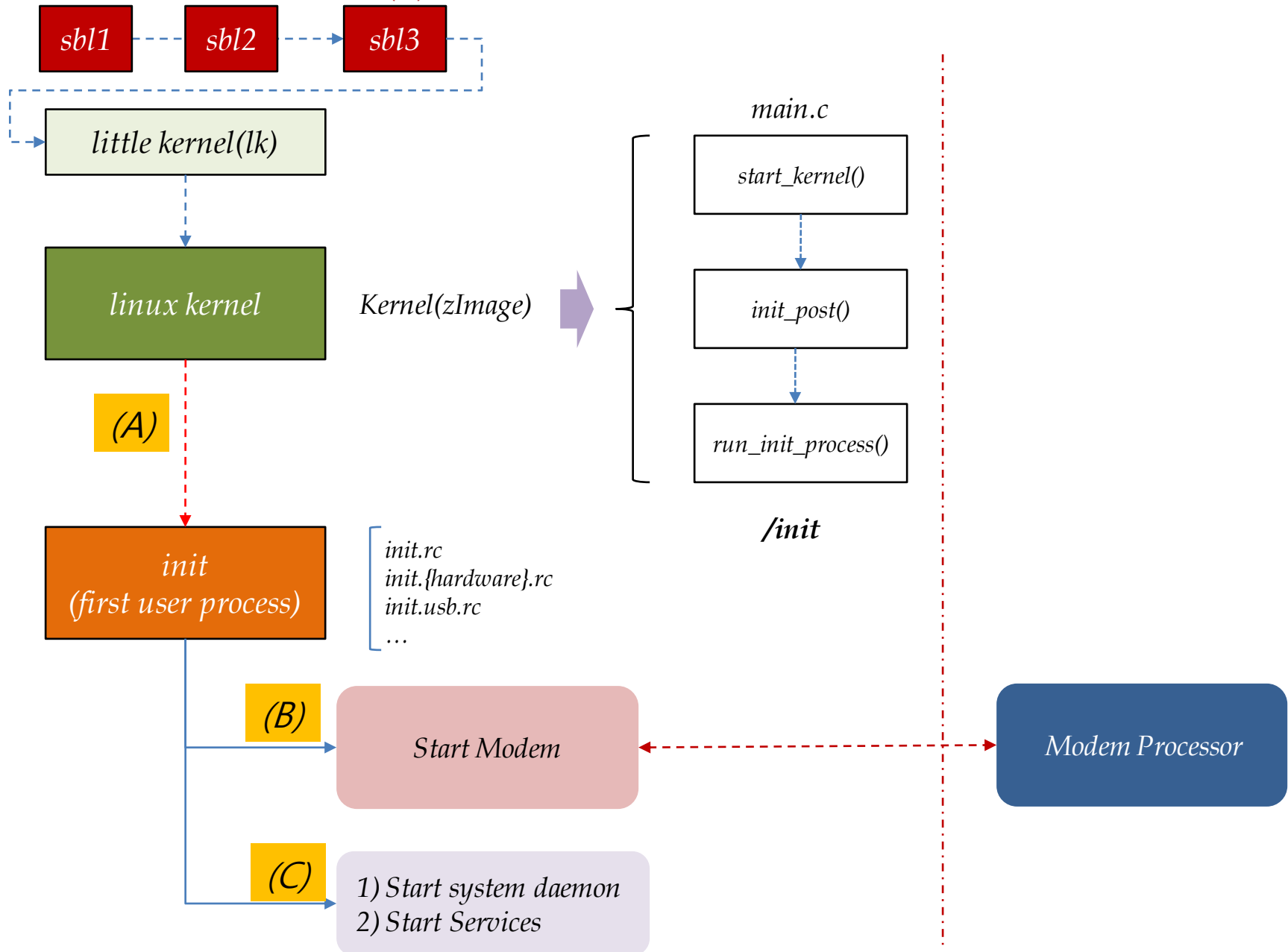
(*) 아래 메모리 map은 개념을 정리하는 차원에서 그려 본 것이며, 실제 내용(memory 번지 등)과는 거리가 있음.

→ 구동 중일 때와 구동 후의 상태를 함께 보여 주고 있음.

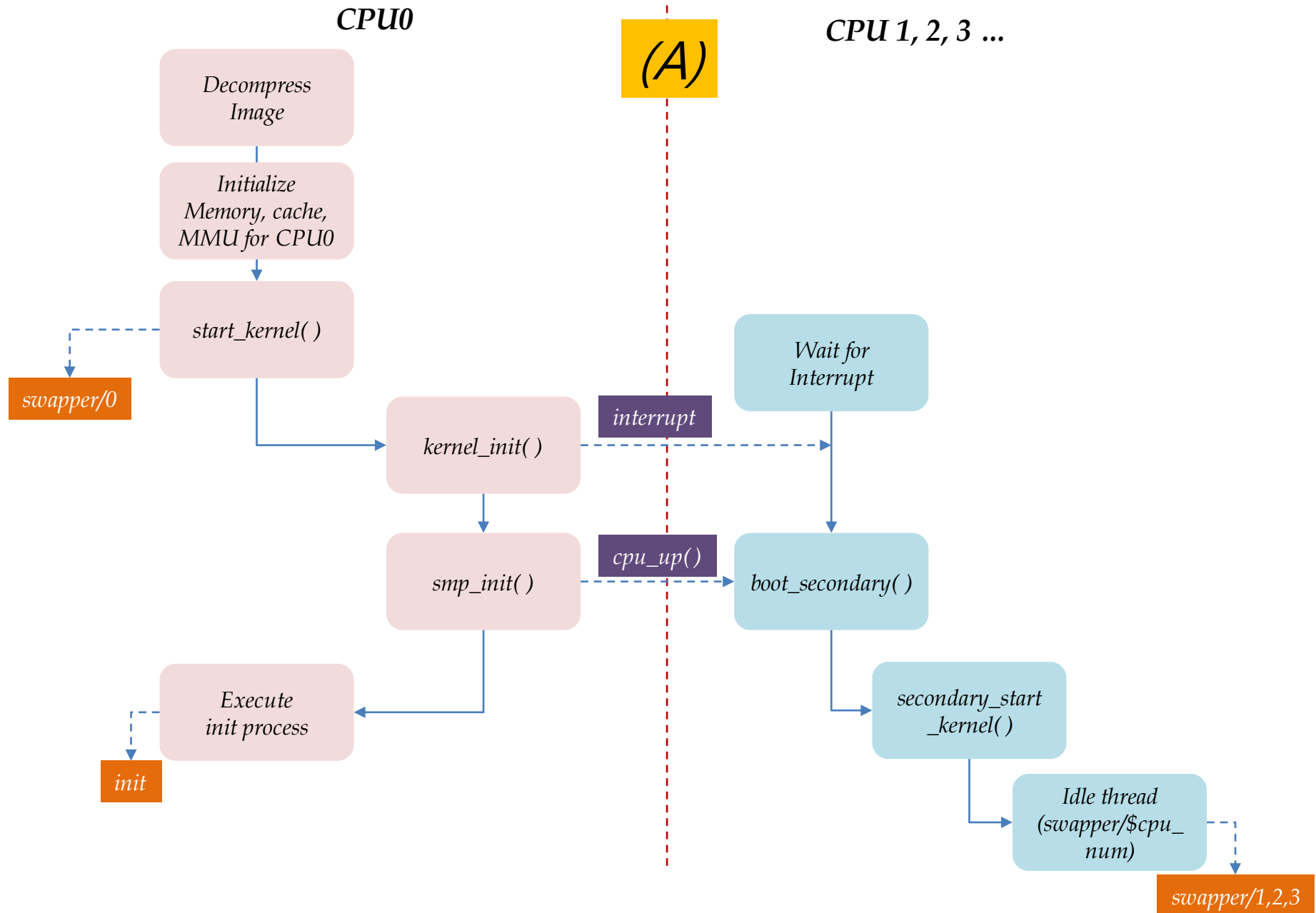
→ Ramdisk 위치는 ATAGS를 통해 bootloader -> kernel로 전달됨.



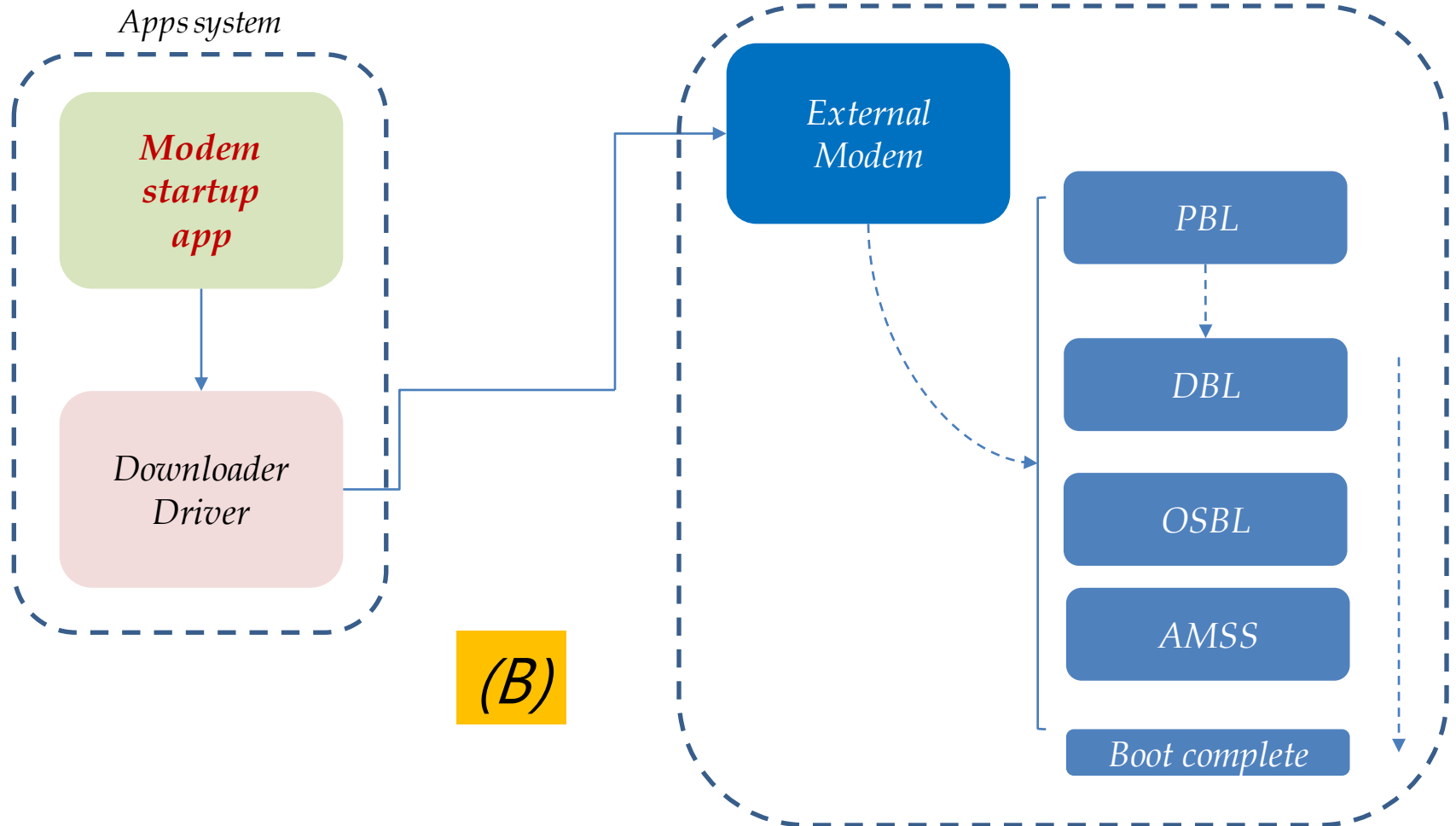
4. Linux Kernel Boot(2)



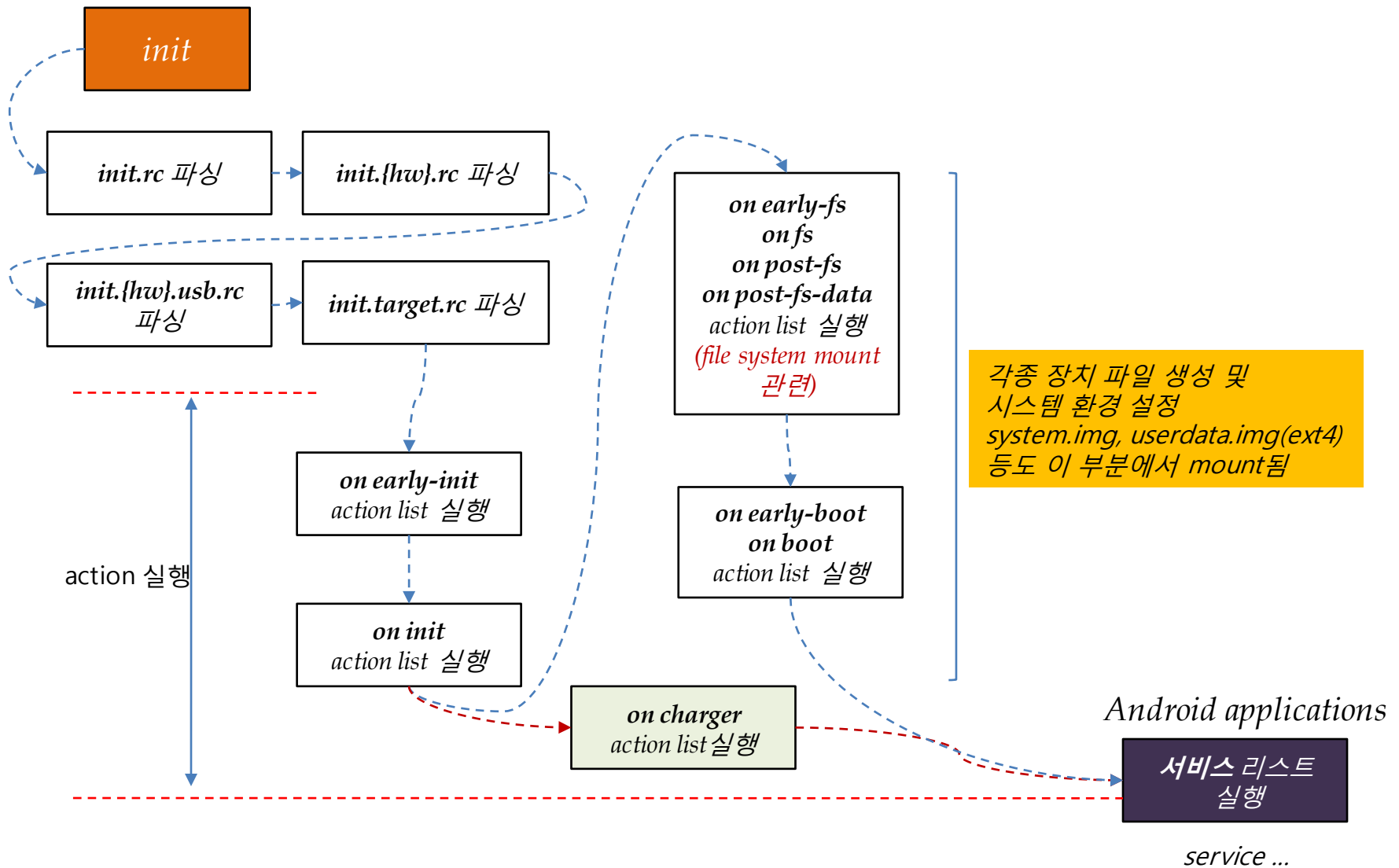
4. Linux Kernel Boot(3) – SMP Boot



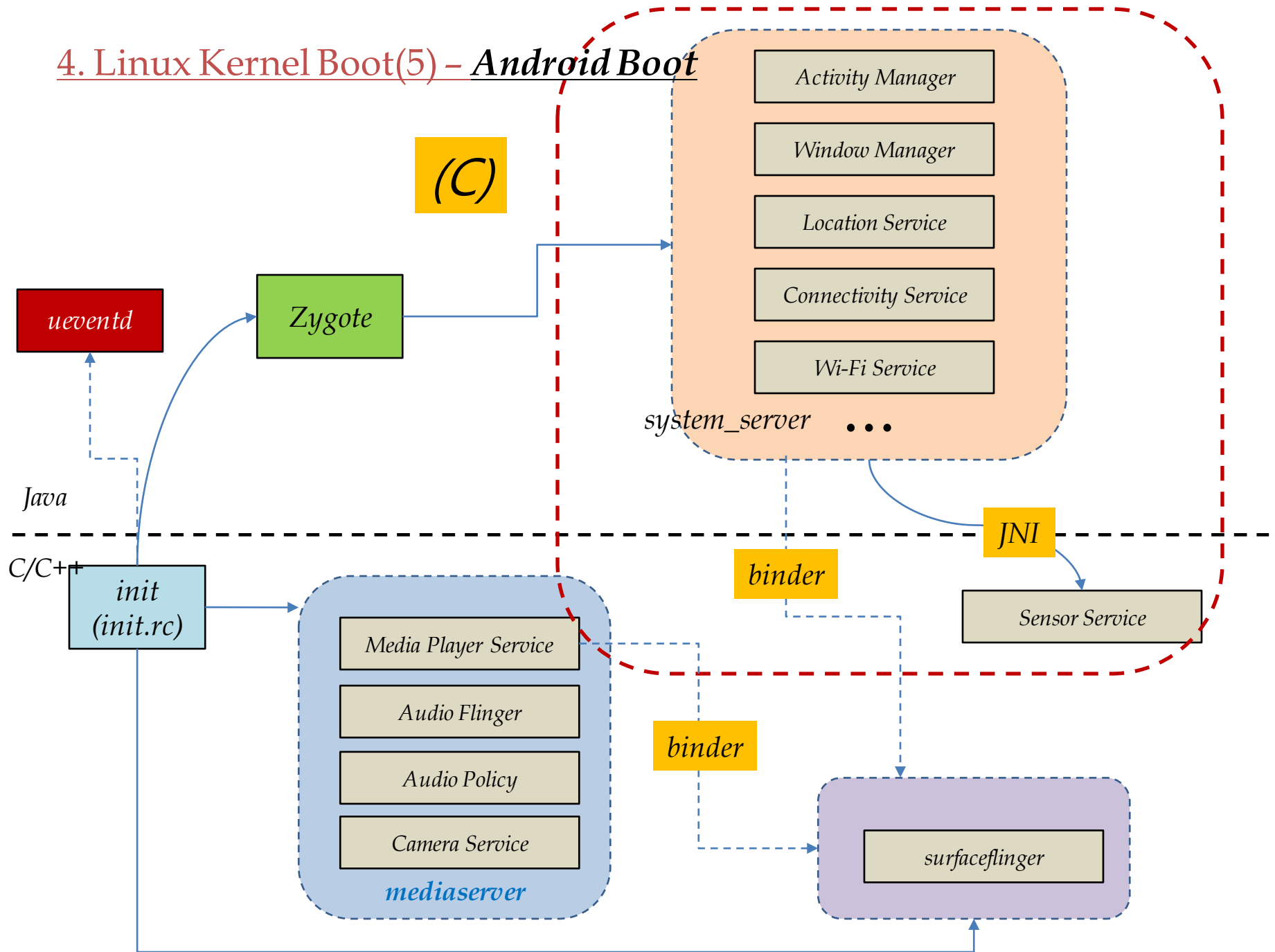
4. Linux Kernel Boot(4) – Modem Loading Example



4. Linux Kernel Boot(5) – *Android Boot(init & init.rc)*

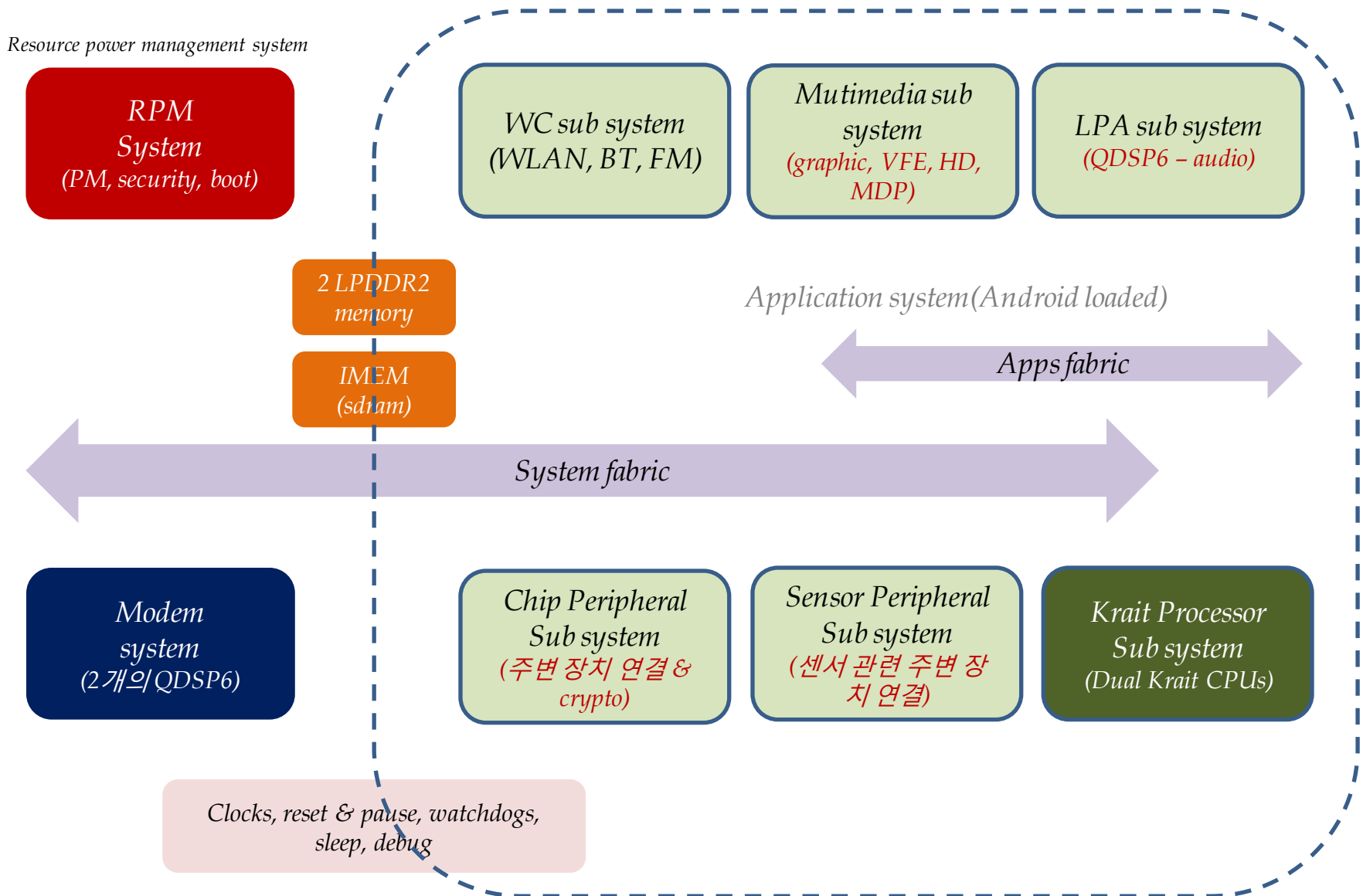


4. Linux Kernel Boot(5) – *Android Boot*



Machine Description

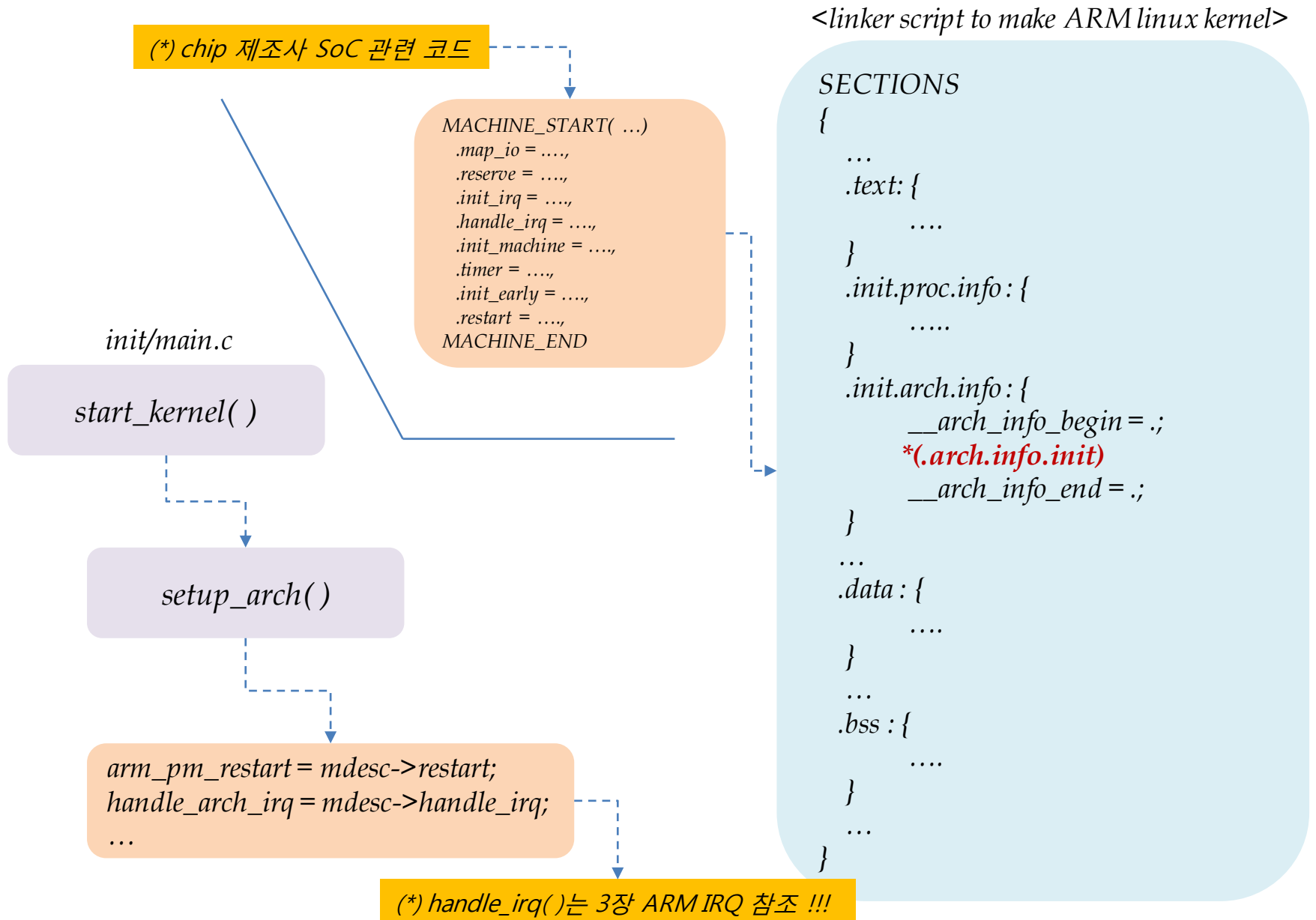
1. MSM8960 Chipset Overview



2. MACHINE_START macro(1) – MSM8960 Example

```
MACHINE_START(MSM8960_CDP, "QCT MSM8960 CDP")
    .map_io = msm8960_map_io,
    .reserve = msm8960_reserve,
    .init_irq = msm8960_init_irq,
    .handle_irq = gic_handle_irq,
    .timer = &msm_timer,
    .init_machine = msm8960_cdp_init,
    .init_early = msm8960_allocate_memory_regions,
    .init_very_early = msm8960_early_memory,
    .restart = msm_restart,
MACHINE_END
```

2. MACHINE_START macro(2) - restart(), handle_irq() 등록



2. MACHINE_START macro(3) – *init_machine()* call

<linker script to make ARM linux kernel>

```
#define INIT_CALLS \
    VMLINUX_SYMBOL(__initcall_start) = .; \
    *(.initcallearly.init) \
    INIT_CALLS_LEVEL(0) \
    INIT_CALLS_LEVEL(1) \
    INIT_CALLS_LEVEL(2) \
    INIT_CALLS_LEVEL(3) \
    INIT_CALLS_LEVEL(4) \
    INIT_CALLS_LEVEL(5) \
    INIT_CALLS_LEVEL(rootfs) \
    INIT_CALLS_LEVEL(6) \
    INIT_CALLS_LEVEL(7) \
    VMLINUX_SYMBOL(__initcall_end) = .;
```

```
SECTIONS
{
    ...
    .init.data : {
        INIT_SETUP(16)
        INIT_CALLS
        CON_INITCALL
        INIT_RAM_FS
    }
    ...
}
```

start_kernel()

rest_init()

kernel_init()

kernel_init_freeable()

do_basic_setup()

do_initcalls()

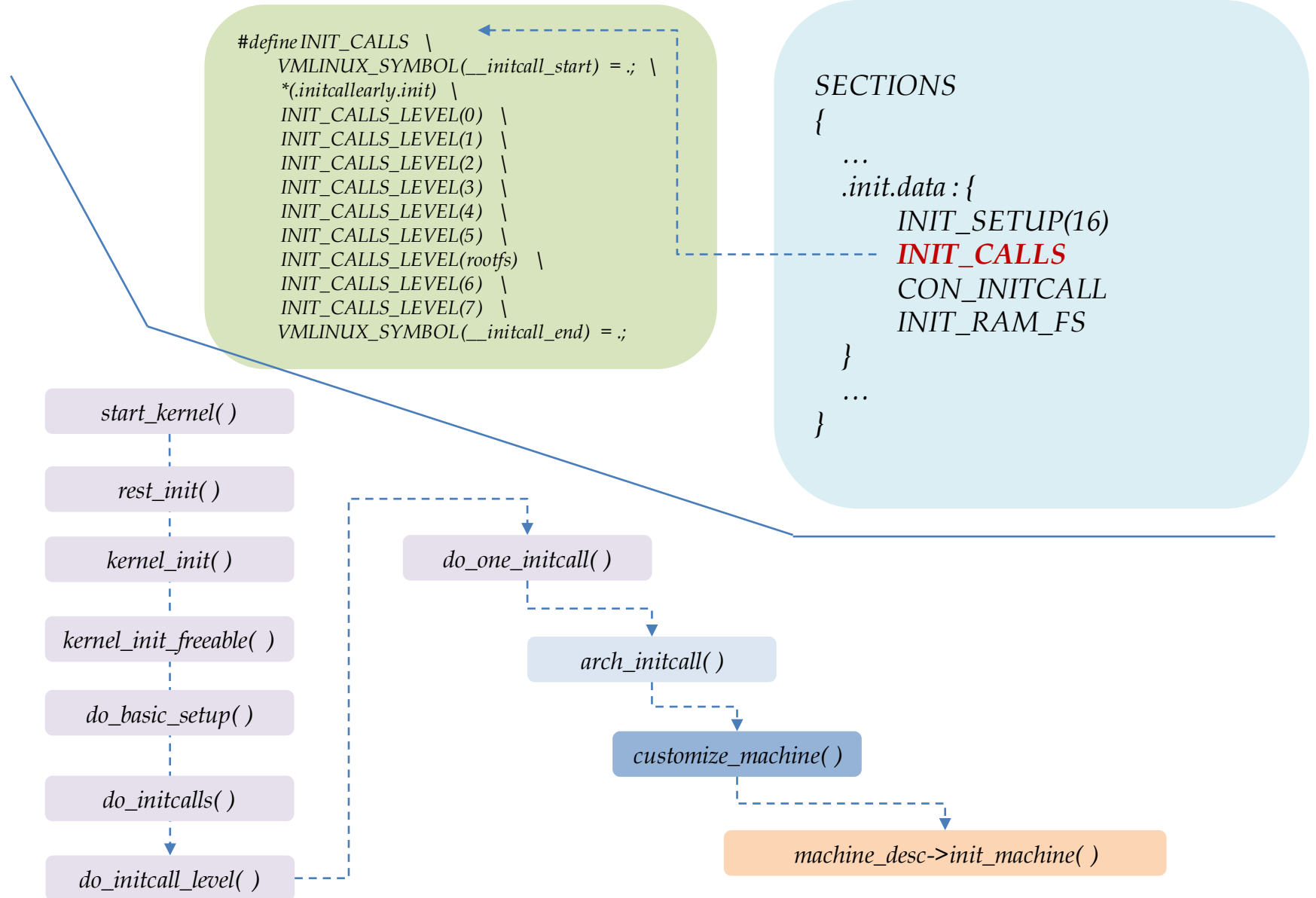
do_initcall_level()

do_one_initcall()

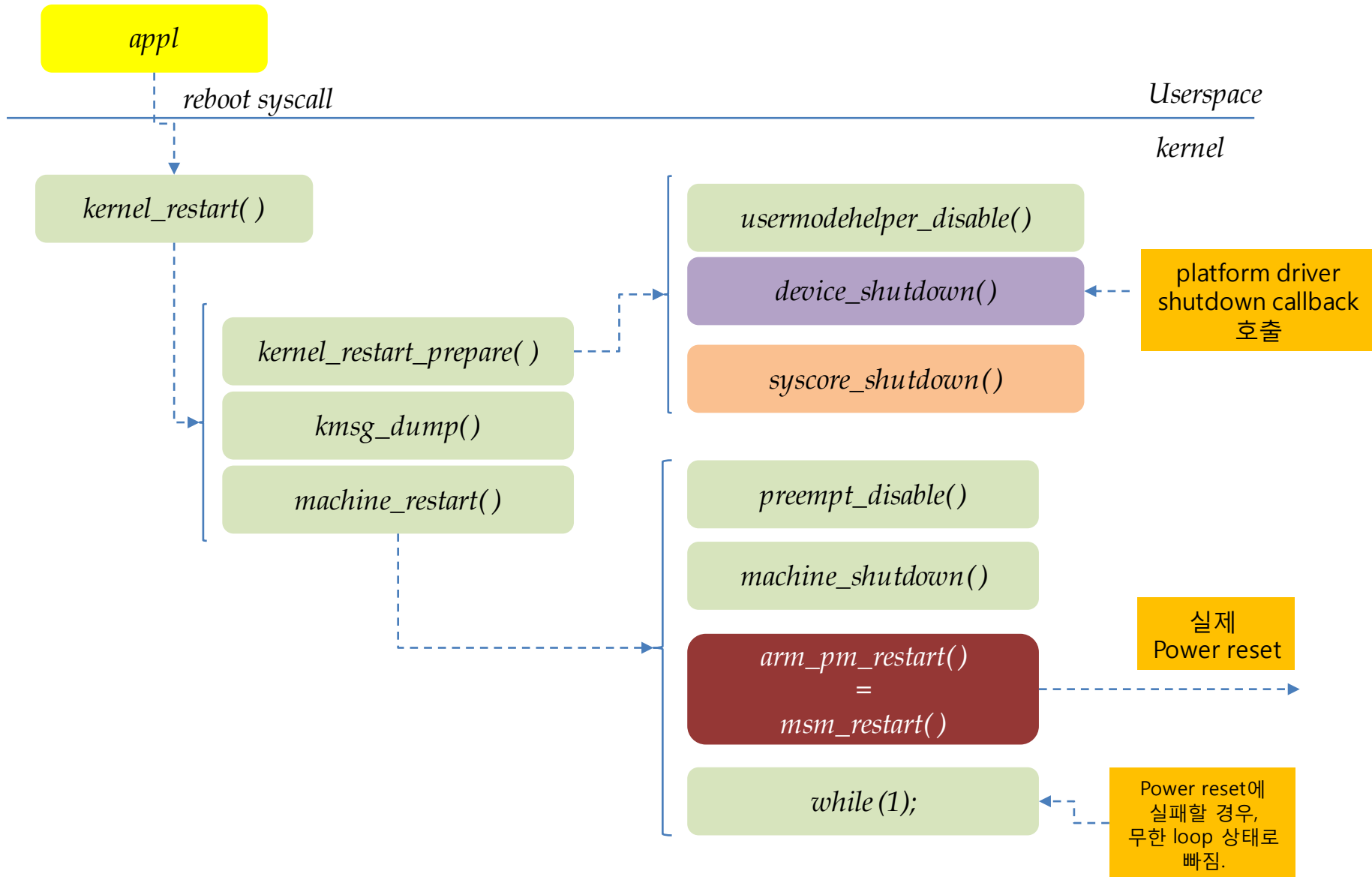
arch_initcall()

customize_machine()

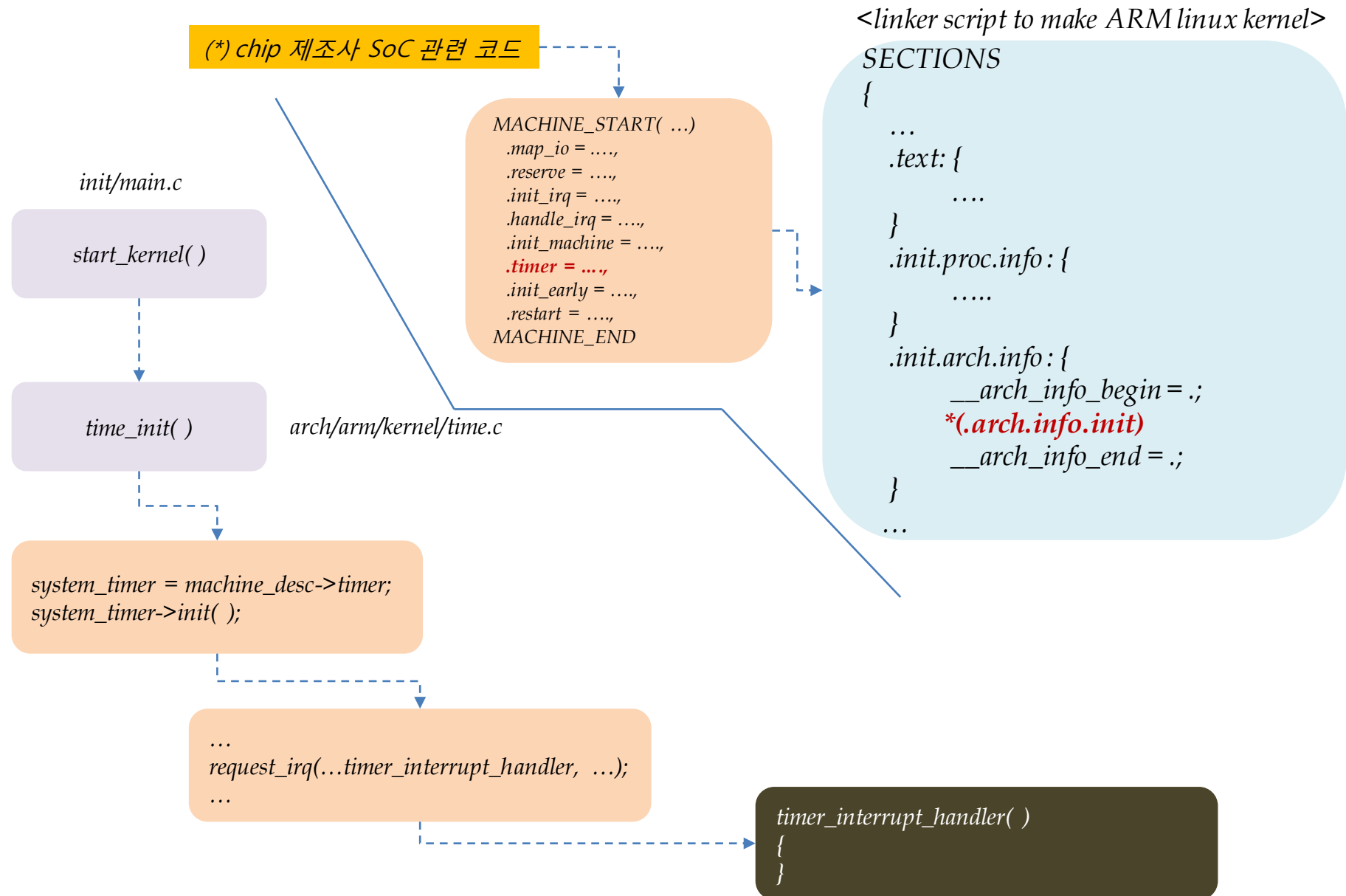
machine_desc->init_machine()



2. MACHINE_START macro(4) – restart() call



2. MACHINE_START macro(5) - *timer()* 등록



2. MACHINE_START macro(6) – *timer interrupt & softirq*

Timer interrupt

softirq_action[]

HI_SOFTIRQ

TIMER_SOFTIRQ

...

(2)

run_timer_softirq()

<timer list 관련 data structure>

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

(1) 신규 타이머 등록 init_timer

timer_list

timer_list

(3) 실행

timer_list

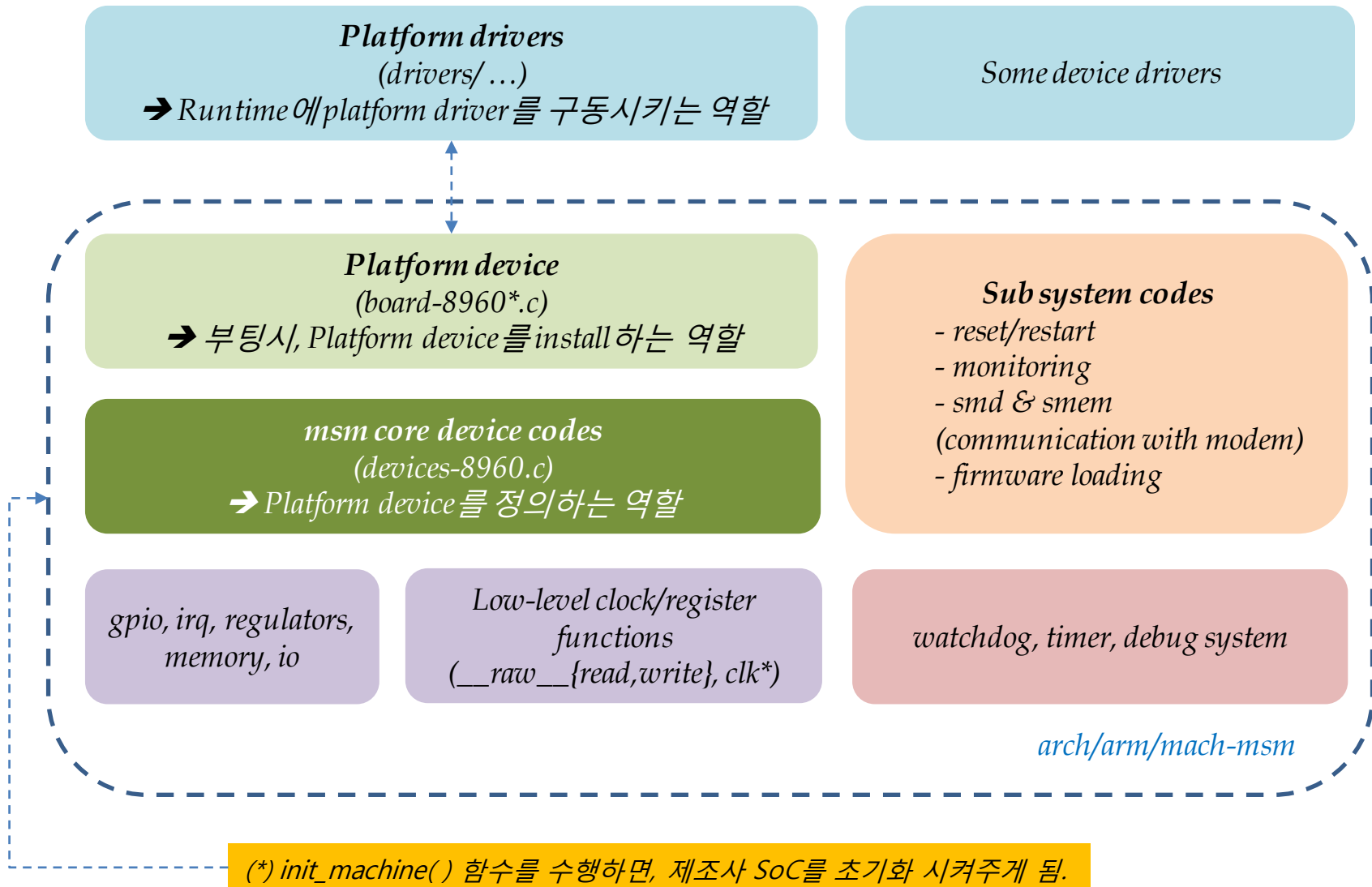
timer_list

timer_list

시간대별로
Queue 관리

timer vector

3. init_machine(1) – MSM8960



3. init_machine(2)

- 1) CPU 기술(description) → clock...
- 2) bus 기술 → system fabric, apps fabric, clocks ...
- 3) memory 기술 & 사용 관련 정의 → pmem, ion mem 등 할당 ...
- 4) CPU와 주변 sub system과의 관계 기술 → interrupt, reset, communication
- 5) 주변 장치 기술/초기화(i2c, spi, uart, uim, sdcc, usb, ..., gpio, irq, regulators) → GSBI & GPIOMux
- 6) key internal functions 기술 → watchdog, sleep...
- 7) linux driver model 적용(platform device & driver)

3. init_machine(3) - 주변 장치 추가 절차(MSM8960 example)

1) Set the clocks

➔ *arch/arm/mach-msm/clock-8960.c*

2) Create a platform device

➔ *arch/arm/mach-msm/devices.h*

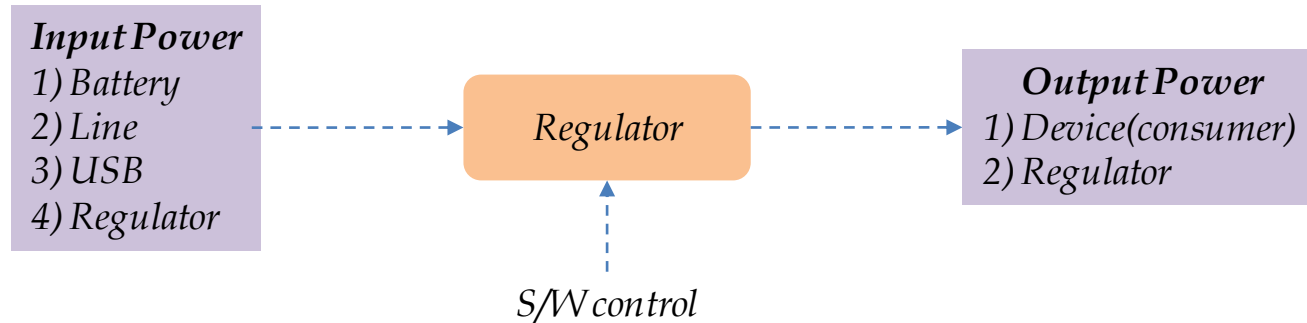
➔ *arch/arm/mach-msm/devices-8960.c*

➔ *arch/arm/mach-msm/board-8960.c*

3) Configure GPIO

➔ *arch/arm/mach-msm/board-8960-gpiomux.c*

3. init_machine(4) – regulator/1



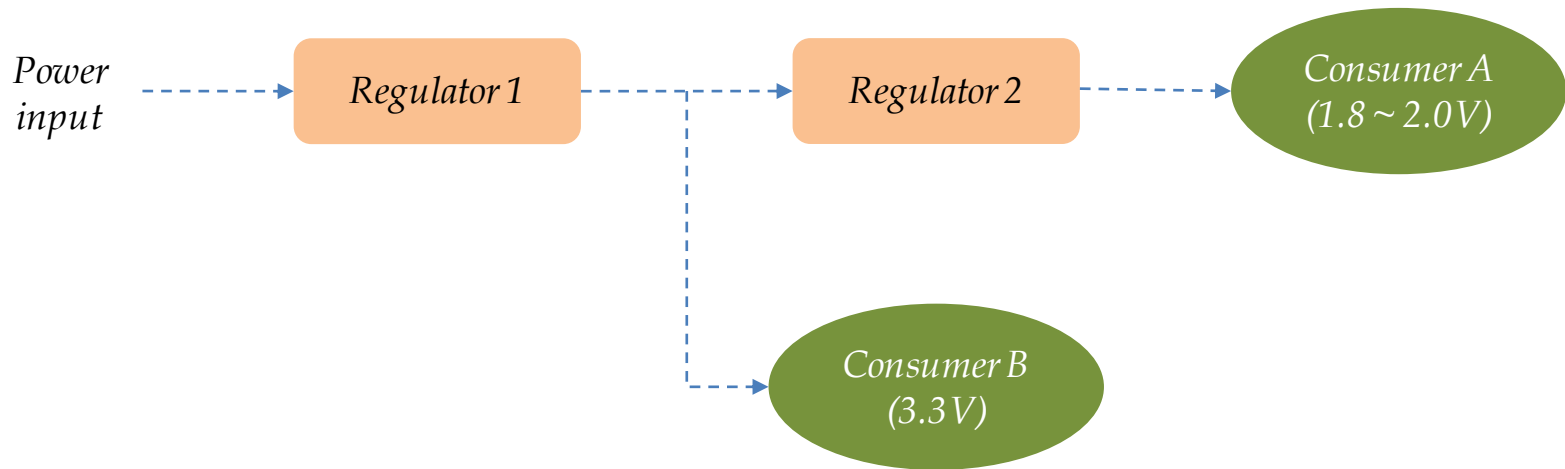
<Regulator의 역할>

- 1) Voltage Control(전압 제어)
- 2) Current Limit(전류 제한)
- 3) Output power on/off

Linux kernel regulator framework

- 1) Machine interface
- 2) Regulator driver interface
- 3) Consumer interface)
- 4) Sysfs interface

3. init_machine(4) – regulator/2



<Step 1>

```
static struct regulator_consumer_supply regulator1_consumers[] = {  
{  
    .dev_name = "dev_name(consumer B)",  
    .supply   = "Vcc",  
},};
```

```
static struct regulator_consumer_supply regulator2_consumers[] = {  
{  
    .dev   = "dev_name(consumer A)",  
    .supply = "Vcc",  
},};
```

다음 페이지 계속

3. init_machine(4) – regulator/3

<Step 2 – Power constraints>

```
static struct regulator_init_data regulator1_data = {
    .constraints = {
        .name = "Regulator-1",
        .min_uV = 3300000,
        .max_uV = 3300000,
        .valid_modes_mask = REGULATOR_MODE_NORMAL,
    },
    .num_consumer_supplies = ARRAY_SIZE(regulator1_consumers),
    .consumer_supplies = regulator1_consumers,
};

static struct regulator_init_data regulator2_data = {
    .supply_regulator = "Regulator-1",
    .constraints = {
        .min_uV = 1800000,
        .max_uV = 2000000,
        .valid_ops_mask = REGULATOR_CHANGE_VOLTAGE,
        .valid_modes_mask = REGULATOR_MODE_NORMAL,
    },
    .num_consumer_supplies = ARRAY_SIZE(regulator2_consumers),
    .consumer_supplies = regulator2_consumers,
};
```

<Step 3 – platform device registration>

```
static struct platform_device regulator_devices[] = {
    {
        .name = "regulator",
        .id = DCDC_1,
        .dev = {
            .platform_data = &regulator1_data,
        },
    },
    {
        .name = "regulator",
        .id = DCDC_2,
        .dev = {
            .platform_data = &regulator2_data,
        },
    },
};

/* register regulator 1 device */
platform_device_register(&regulator_devices[0]);

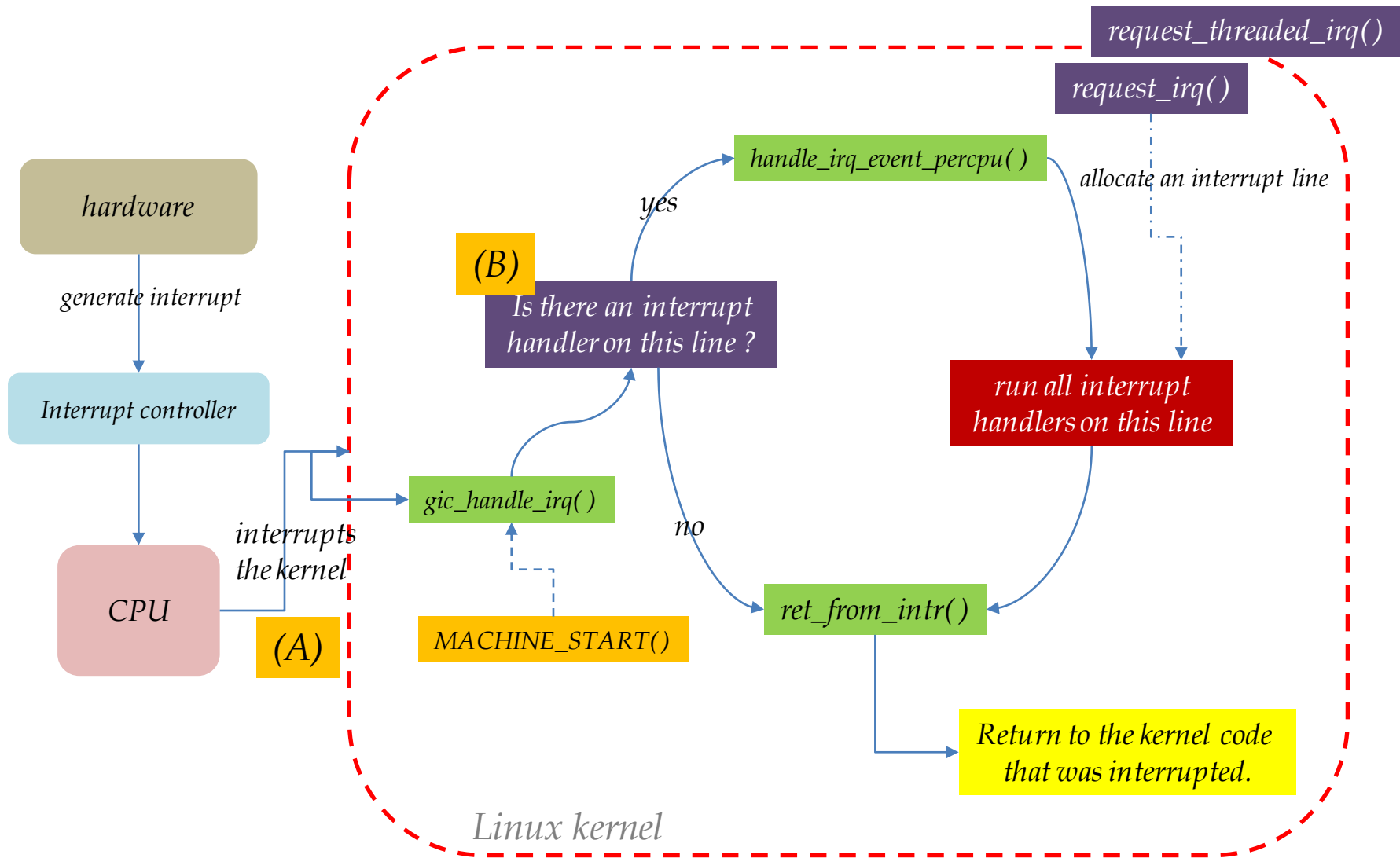
/* register regulator 2 device */
platform_device_register(&regulator_devices[1]);
```

4. init_machine(5) - TODO

- *gpio(mux)*
- *clock*
- ...

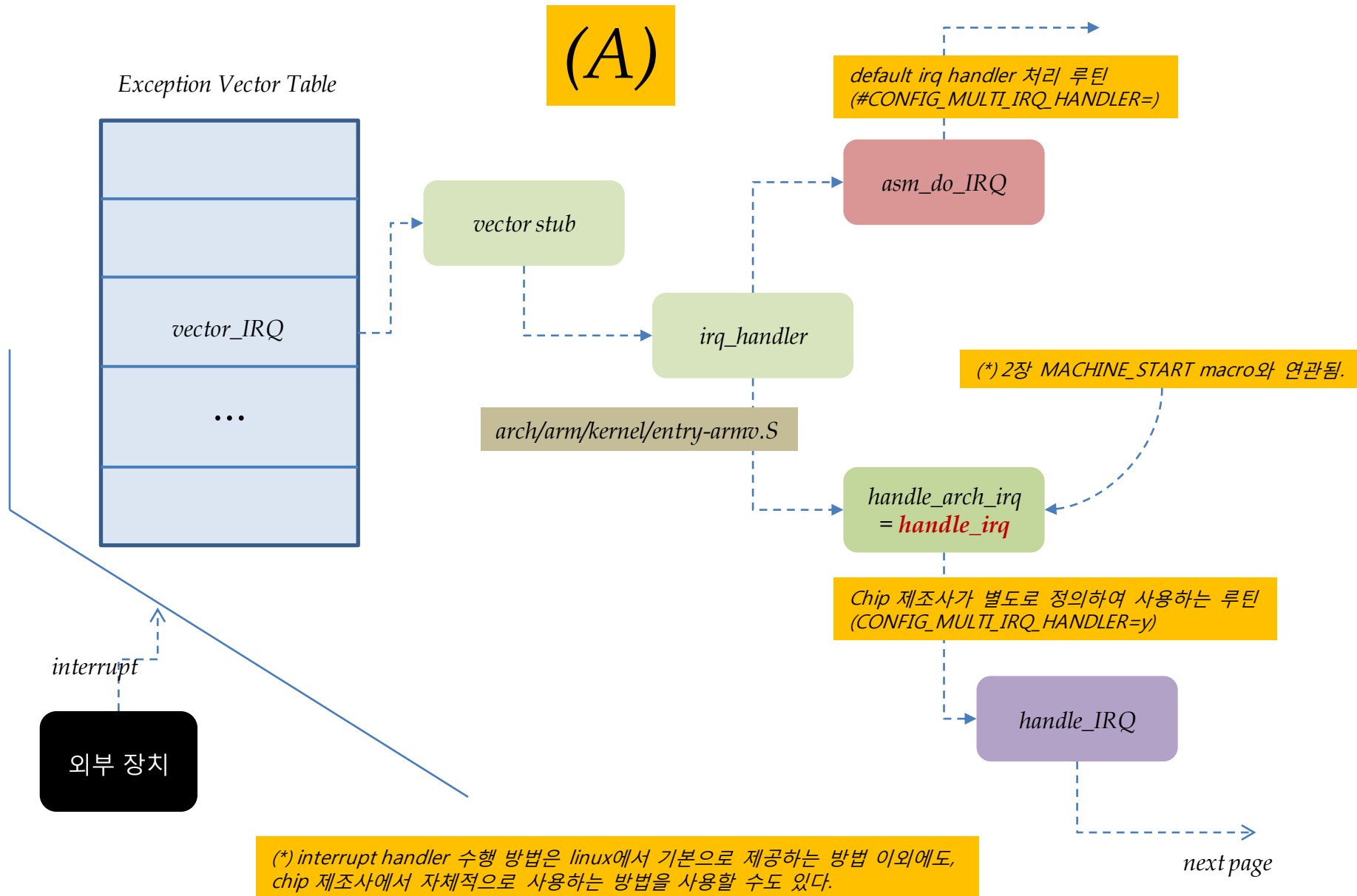
ARM IRQ

1. Interrupt Handling Code Review(1)

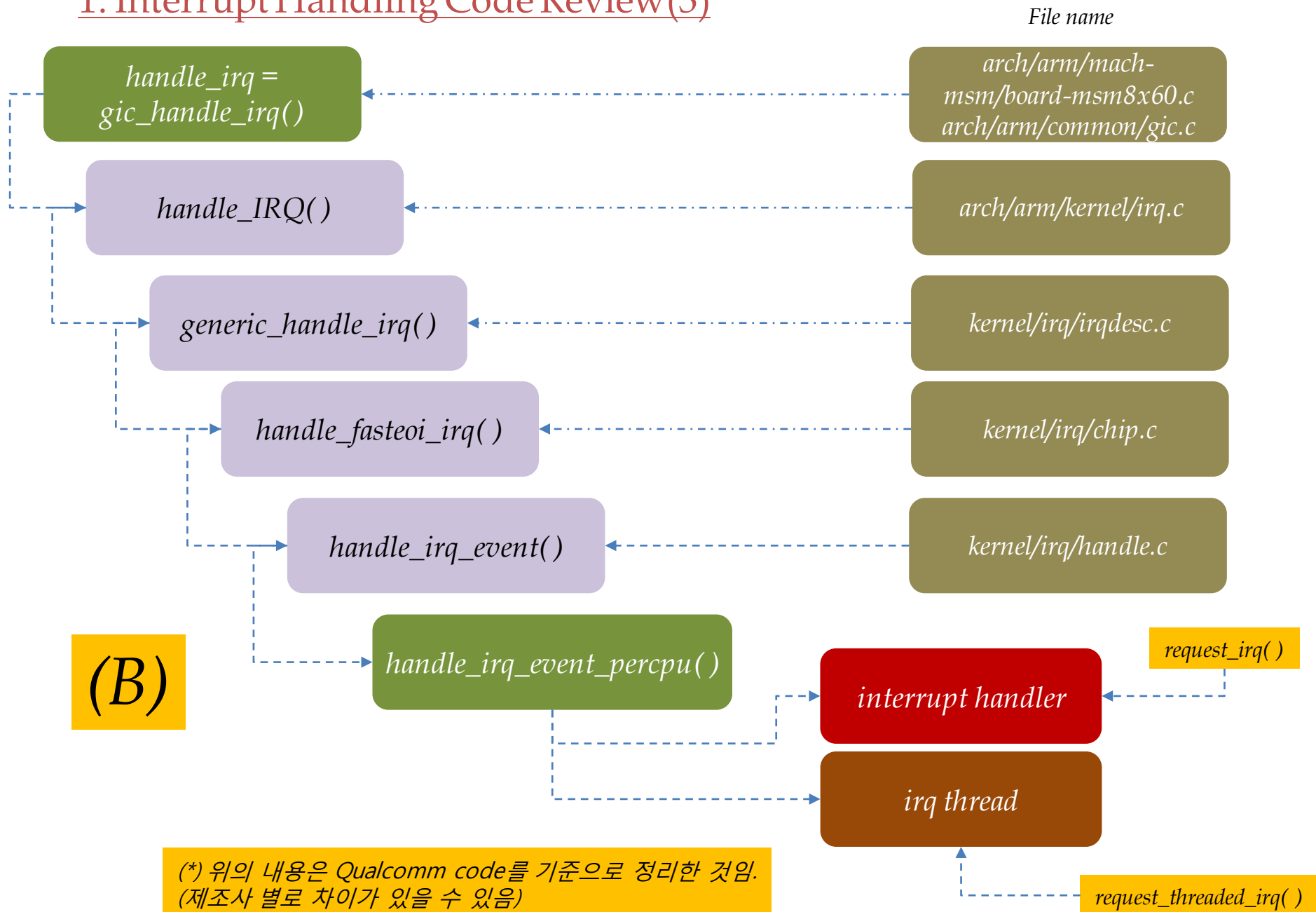


(A), (B) 는 다음 page에서 자세히 설명함.

1. Interrupt Handling Code Review(2)



1. Interrupt Handling Code Review(3)



1. Interrupt Handling Code Review(4)

```
handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)
{
    do {
        res = action->handler(...); /* irq handler 실행 */
                                   /* request_irq, request_threaded_irq로 등록 */
        WARN_ONCE(!irqs_disabled(), ...);
        /* → local irq가 disable되어 있지 않으면, warning message 출력 */
        /* interrupt 처리 중에는 다른 interrupt가 들어오면 안됨. */

        if (res == IRQ_WAKE_THREAD) /* interrupt thread 이면)
            irq_wake_thread(irq, action) /* thread를 깨워줌 */

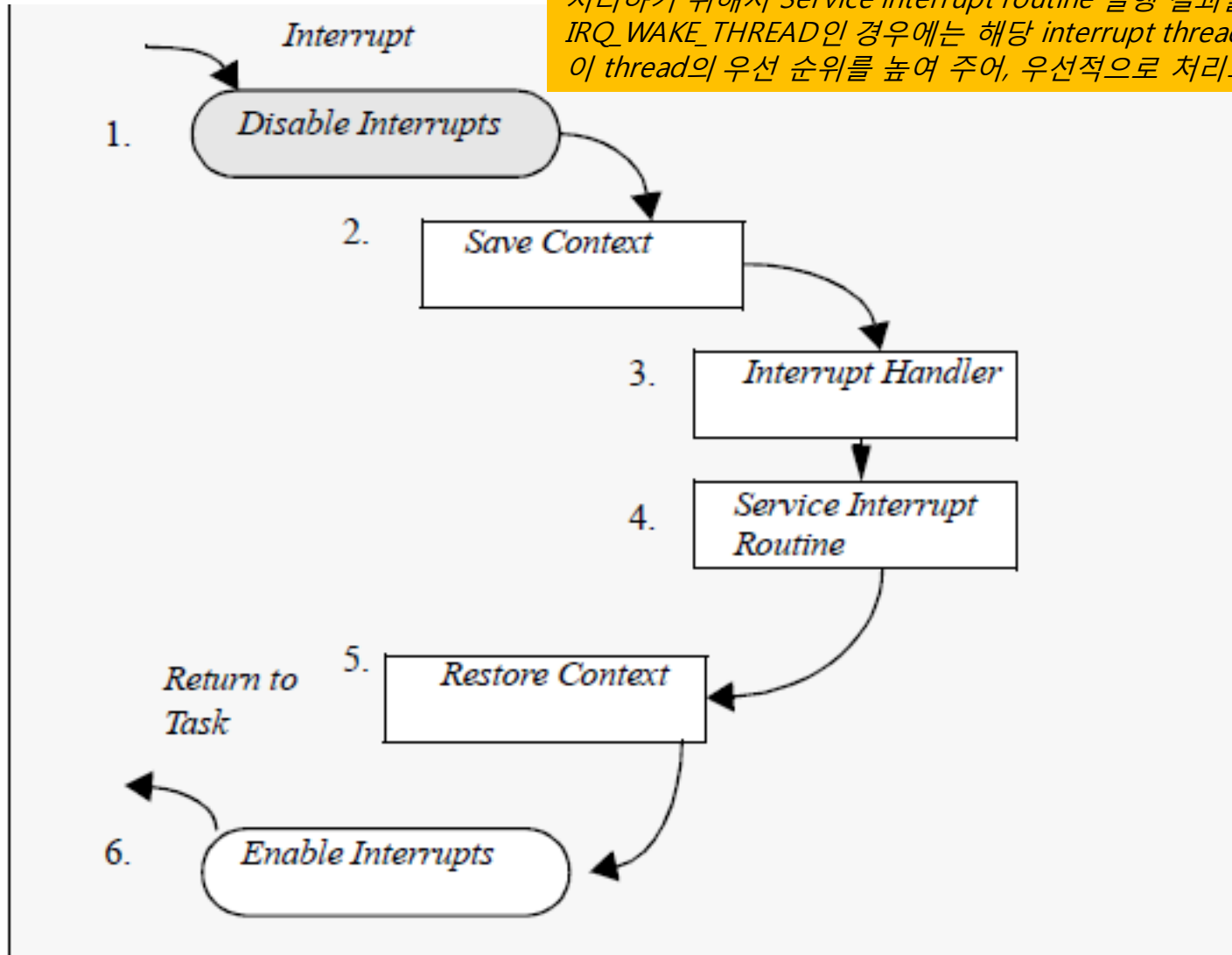
        action = action->next;          /* 다음 action 선택 */
                                       /* 같은 interrupt line, 복수개의
                                       handler 등록 시 사용 */

    } while (action);

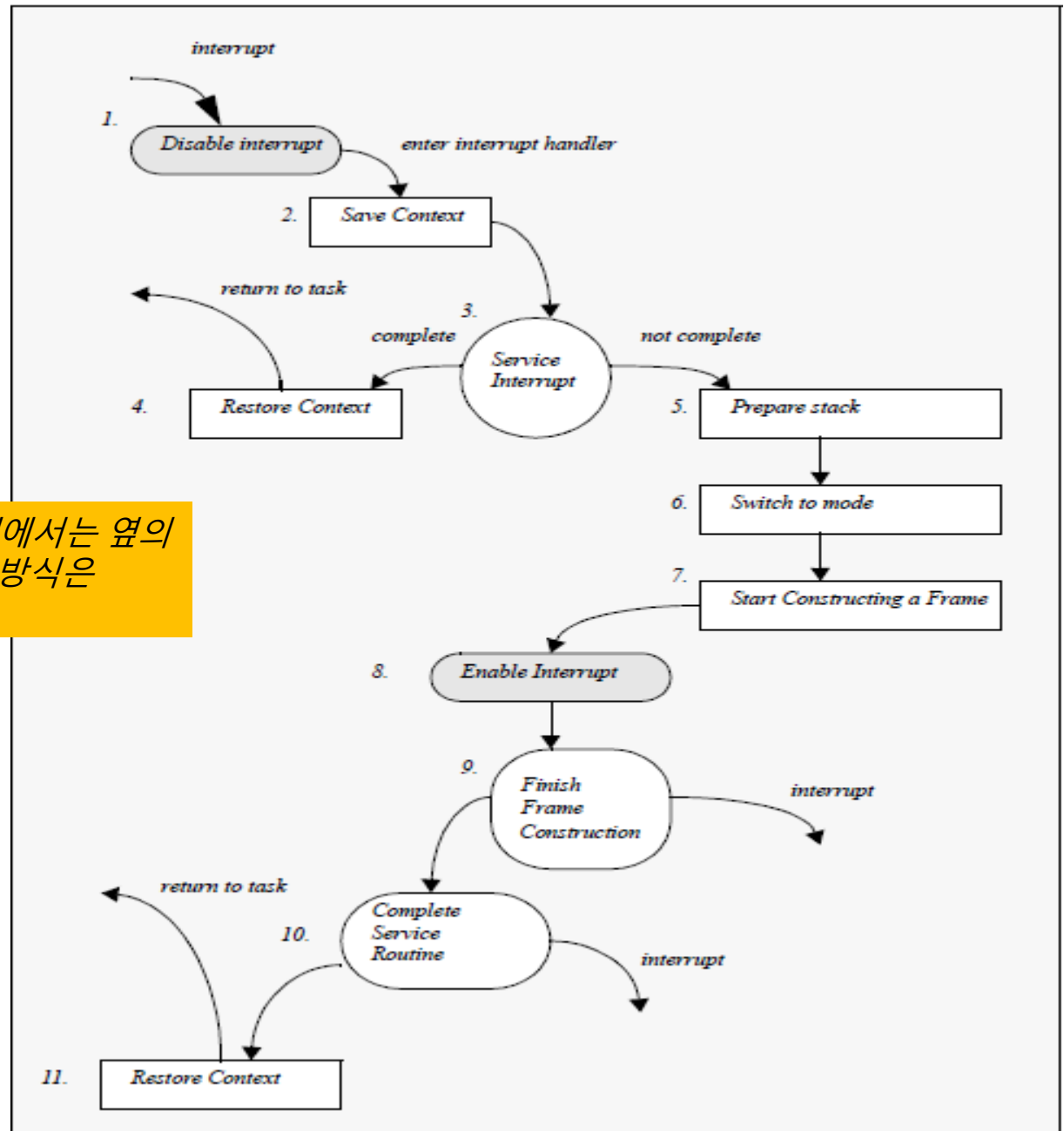
    ...
}
```

2. Interrupt Handler(1) - Non-nested Interrupt Handler

(*) linux 3.8.x(for ARM)에서는 아래의 non-nested interrupt 방식을 지원함.
다만, 동일한 구조를 유지한 상태에서 우선 순위가 높은 interrupt를
처리하기 위해서 Service interrupt routine 실행 결과를 토대로,
IRQ_WAKE_THREAD인 경우에는 해당 interrupt thread를 깨워 주게 되고,
이 thread의 우선 순위를 높여 주어, 우선적으로 처리되는 방식을 취한다^^

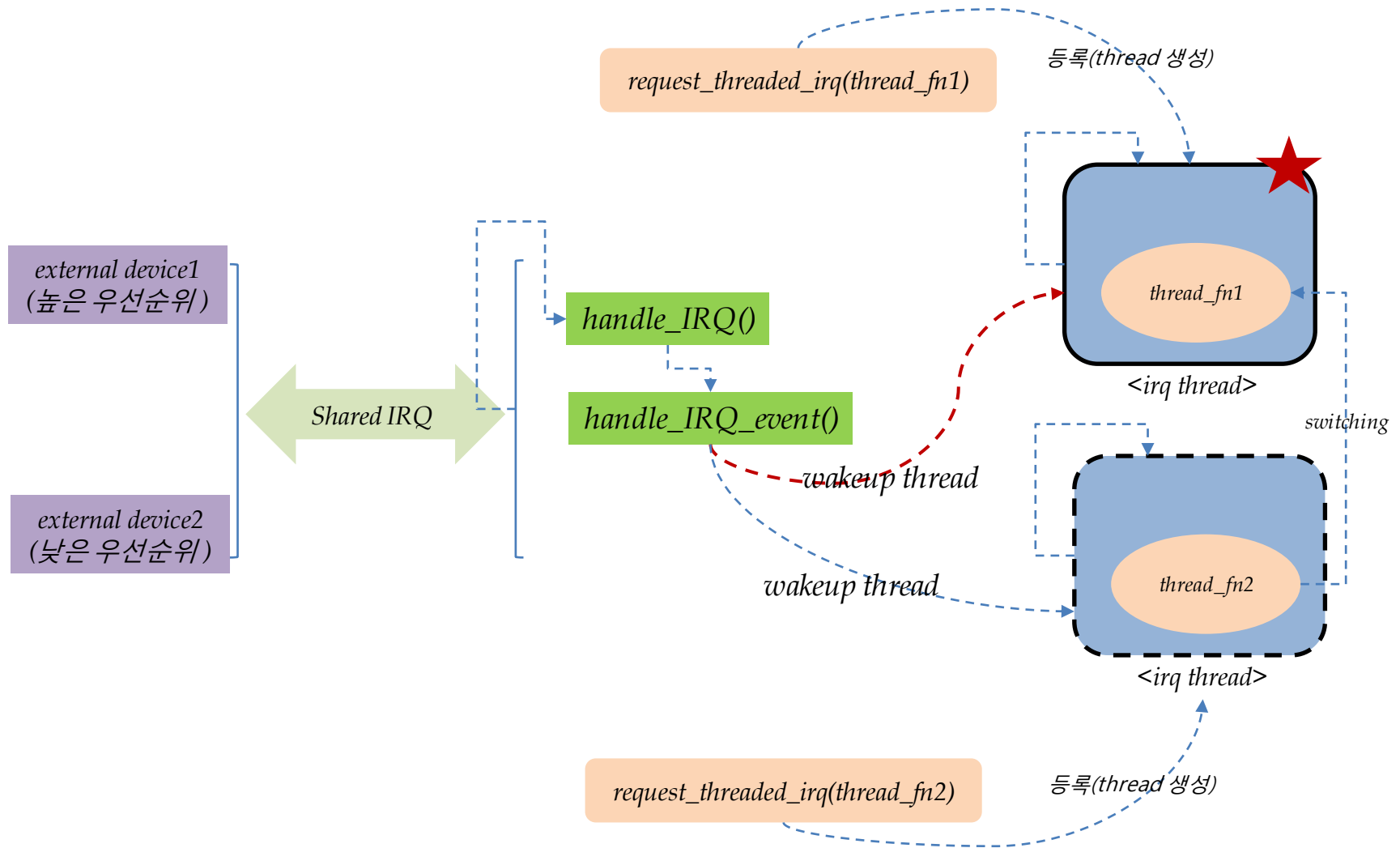


2. Interrupt Handler(2) – Nested interrupt handler



linux 3.8.x(for ARM) kernel에서는 옆의
nested interrupt handling 방식은
지원하지 않음.

2. Interrupt Handler(3) – Shared IRQ/1



(*) Shared IRQ 방식을 사용하게 되면, 같은 interrupt line을 사용하면서 서로 다른 여러 개의 interrupt handler 등록이 가능하다.

(*) 또한, 동일한 interrupt handler를 여러 개 등록할 수도 있다.

2. Interrupt Handler(3) – Shared IRQ/2

<Shared handler 구현 시 요구 사항>

*) 한 개의 *interrupt line*을 여러 장치가 공유(따라서, *interrupt handler*도 각각 서로 다름)할 경우에는 좀 더 특별한 처리가 요구된다.

1) *request_irq()*함수의 *flags* 인자로 *IRQF_SHARED*를 넘겨야 한다.

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

3) 마지막으로 *interrupt handler*는 자신의 *device*가 실제로 *interrupt*를 발생시켰는지를 판단할 수 있어야 한다. 이를 위해서는 *handler* 자체만으로는 불가능하므로, *device*에서도 위를 위한 방법을 제공해야 하며, *handler*도 이를 확인하는 루틴을 제공해야 한다.

*Kernel*이 *interrupt*를 받으면, 등록된 모든 *interrupt handler*를 순차적으로 실행하게 된다. 따라서, *interrupt handler* 입장에서는 자신의 *device*로 부터 *interrupt*가 발생했는지를 판단하여, 그렇지 않을 경우에는 재빨리 *handler* 루틴을 끝내야 한다.

3. Interrupt Handler 등록(1) - `request_irq(=hardirq)`

```
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(첫 번째 argument)`가 interrupt number 임.

<두 번째 argument handler>
`typedef irqreturn_t (*irq_handler_t)(int, void *);`

(*) `/proc/interrupts`에서 인터럽트 상태를 확인할 수 있음 !

H/W interrupt가 발생할 때마다 호출됨

Interrupt handler

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

`synchronize_irq()`

➔ `free_irq`를 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.

3. Interrupt Handler 등록(2) - request threaded irq

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

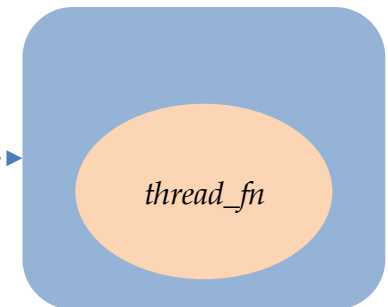
→ Interrupt handler & threaded interrupt handler
등록 및 실행 요청

→ 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 thread>



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

3. Interrupt Handler 등록(3) - *request_any_context_irq*

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

- Context 내역을 보고, hardirq 혹은 threaded irq를 선택
- Return 값: IRQ_IS_NESTED, IRQ_IS_HARDIRQ,

Context를 보고, hardirq로 처리할지,
Threaded irq로 처리할지 분기함.

→
`struct irq_desc *desc = irq_to_desc(irq);`

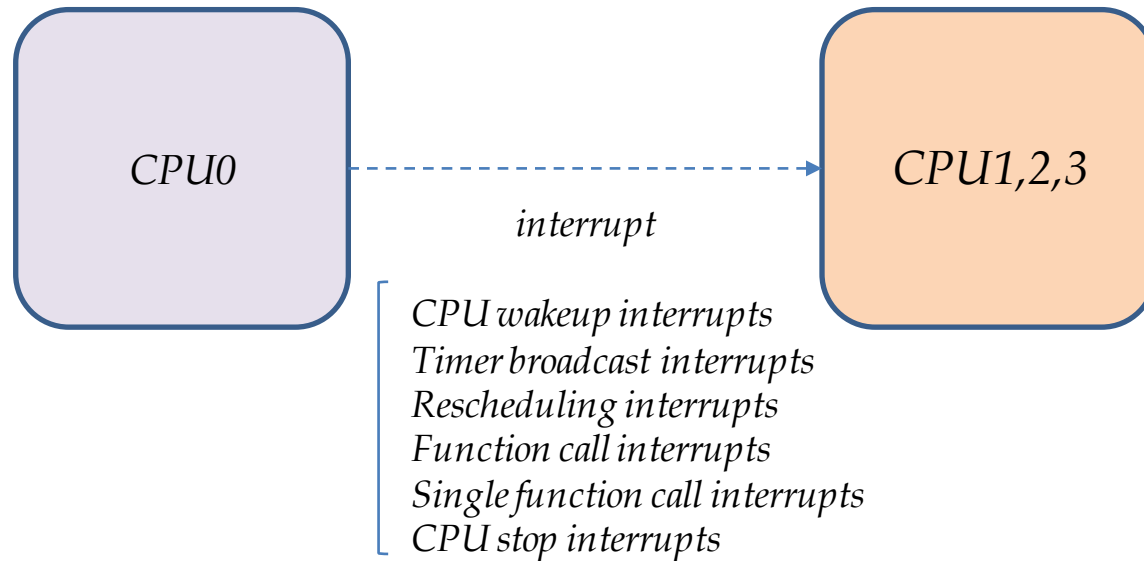
hardirq
handler

<irq thread>

thread_fn

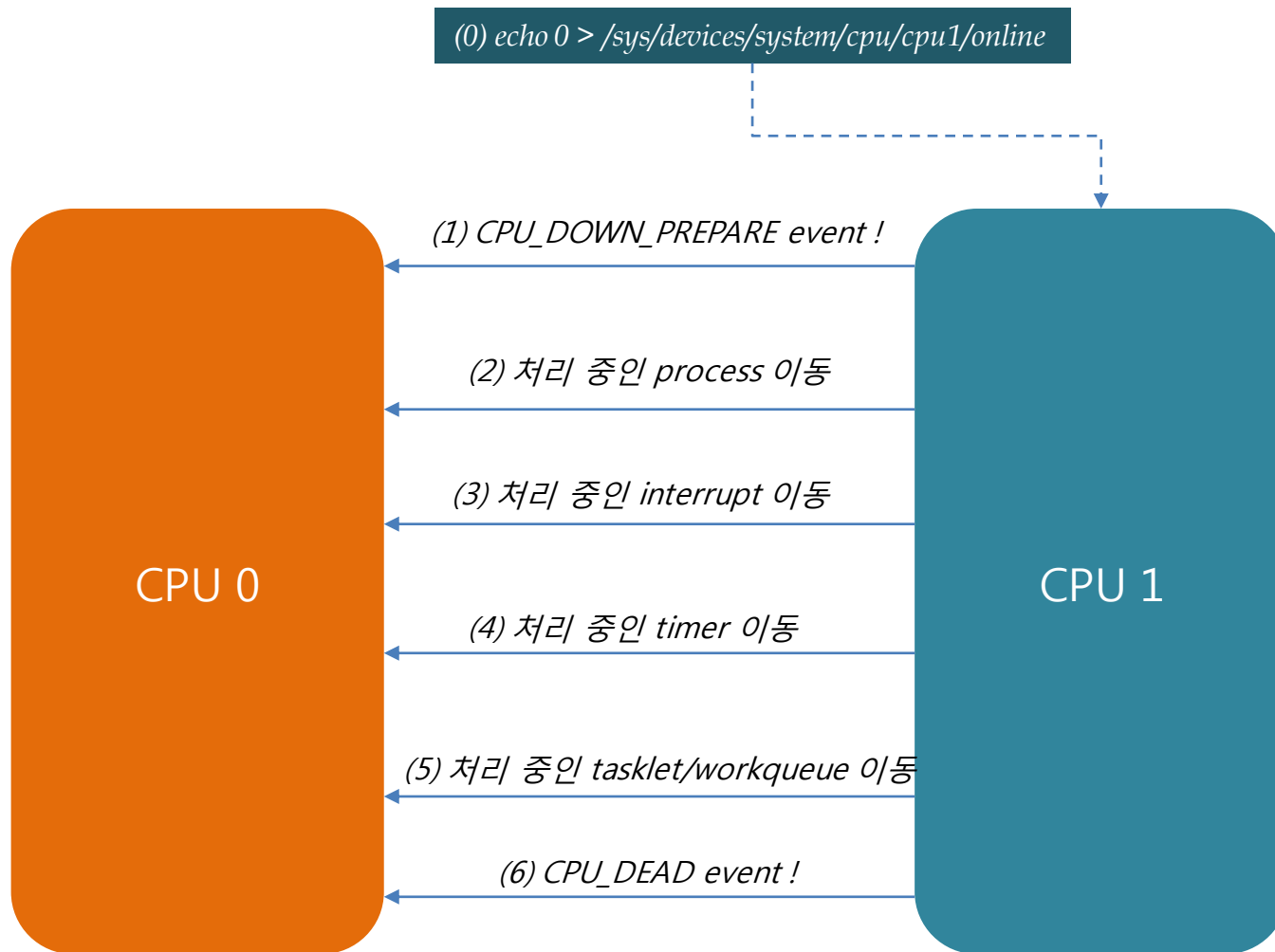
irq/number-name
형태로 thread명칭이
생성됨.
(예: irq/11-myirq)

4. IPI(Inter-Processor Interrupt)



CPU Hotplug

1. CPU Hotplug(1) – *cpu1 down flow*



1. CPU Hotplug(2) – *workqueue* 이동 관련 *event(notifier)* 전송

0) *notifier callback* 정의

```
workqueue_cpu_up_callback(...action...)
{
    switch(action) {
        case CPU_UP_PREPARE:
            ...
            break;

        case CPU_DOWNFAILED:
        case CPU_ONLINE:
            break;
    }
    return NOTIFY_OK;
}

...
```

cpu up or down 등의 *event* 발생 시, *notifier_call* 호출 !

1) *notifier block* 정의 & 등록

```
struct notifier_block
workqueue_cpu_up_callback_nb = {
    .notifier_call = workqueue_cpu_up_callback,
    .priority = CPU_PRI_WORKQUEUE_UP
};

register_cpu_notifier(&cpu_up_callback_nb );
```

kernel/workqueue.c

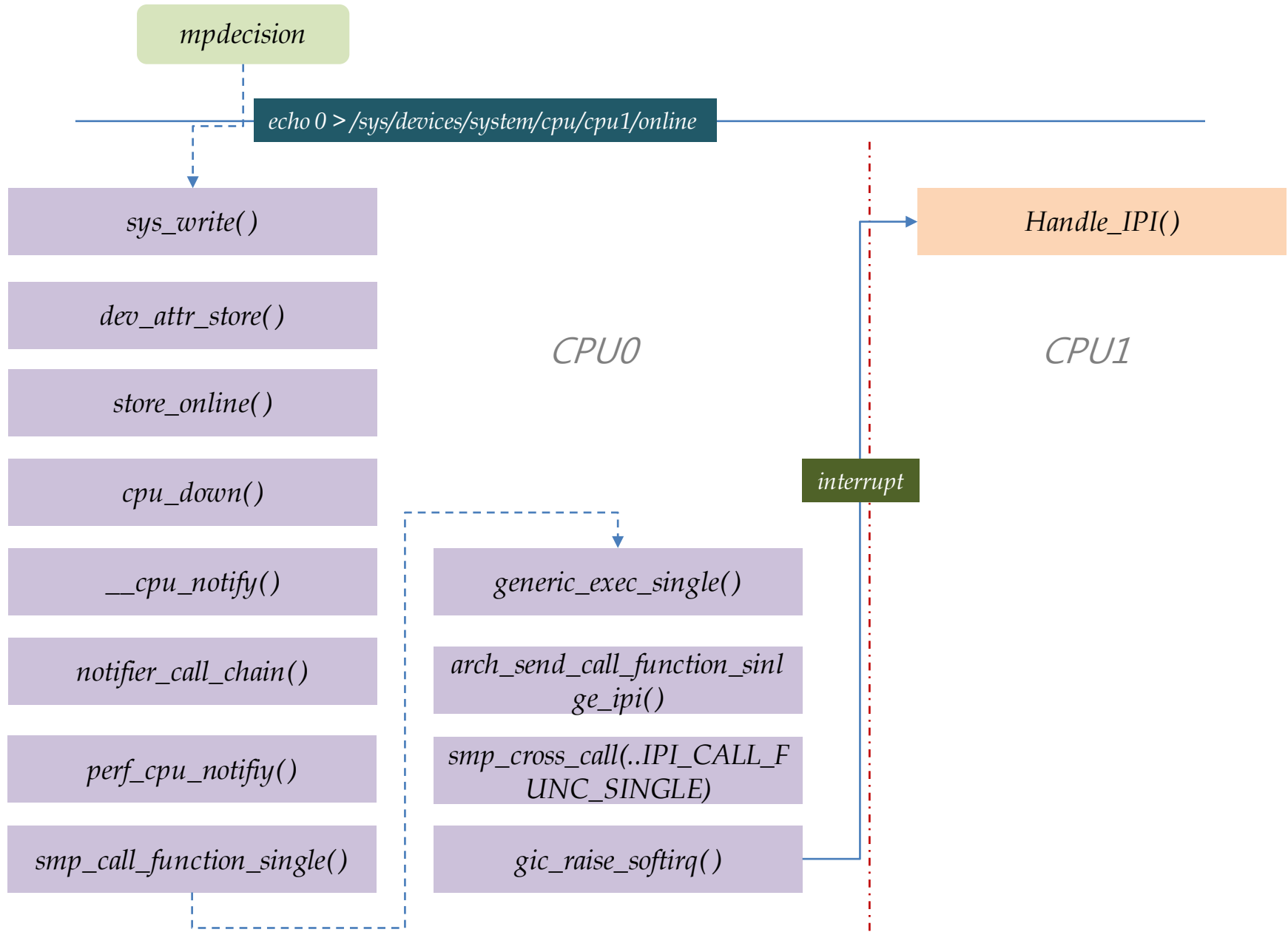
2) *notifier chain* 호출

```
__cpu_notify(...)
{
    ...

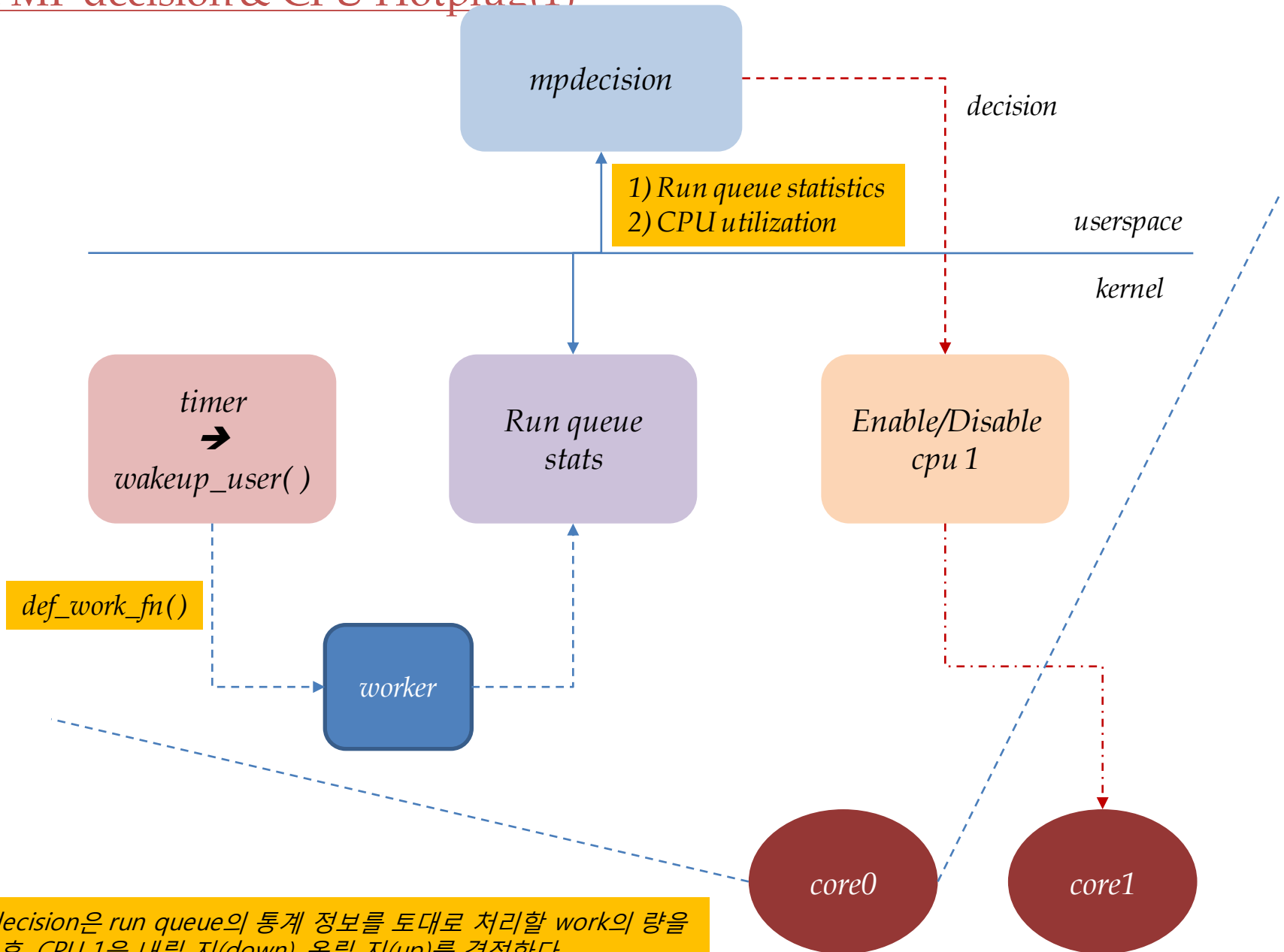
    __raw_notifier_call_chain( ... );
    ...
}
```

kernel/cpu.c

1. CPU Hotplug(3) – function call flow

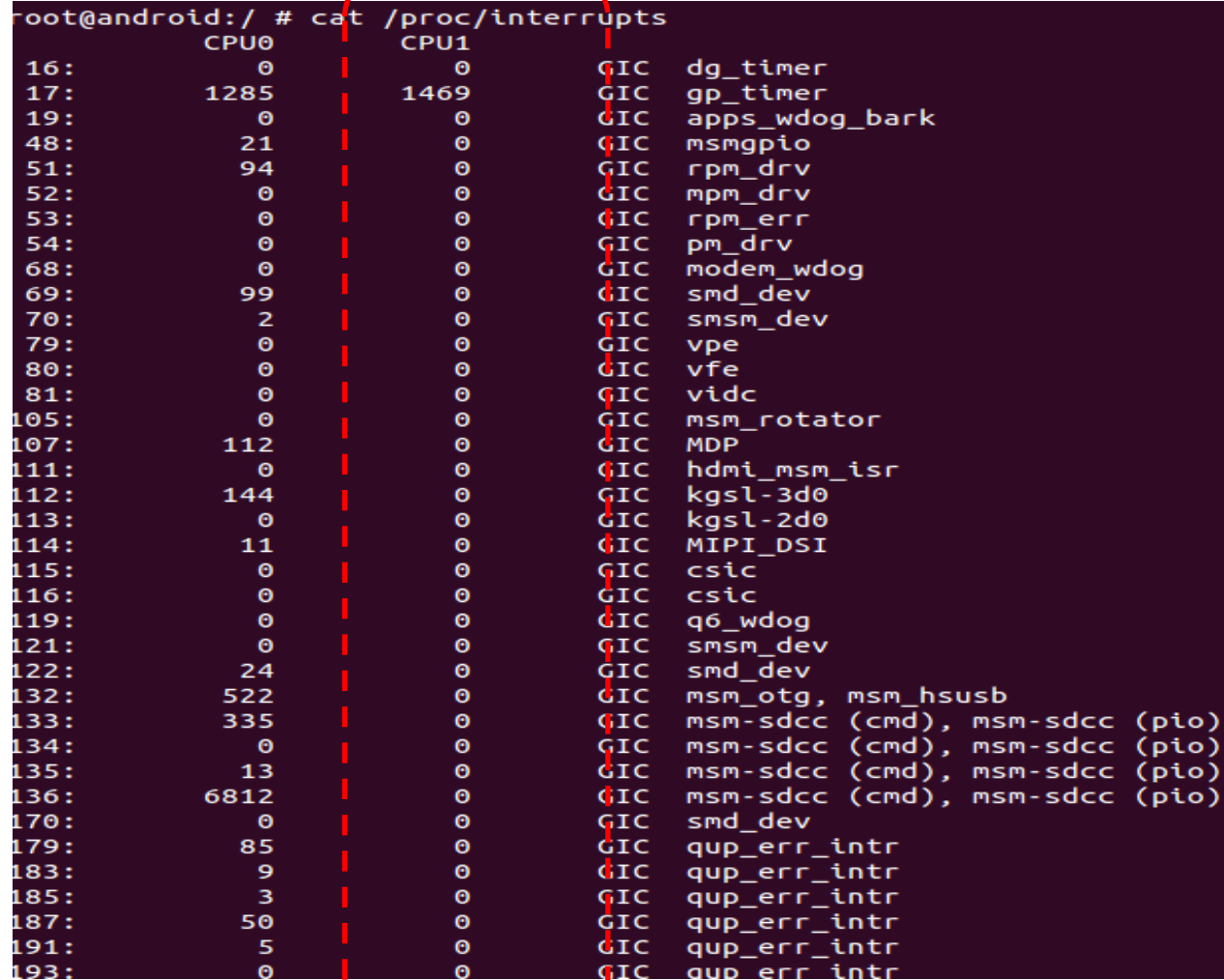


2. MP decision & CPU Hotplug(1)



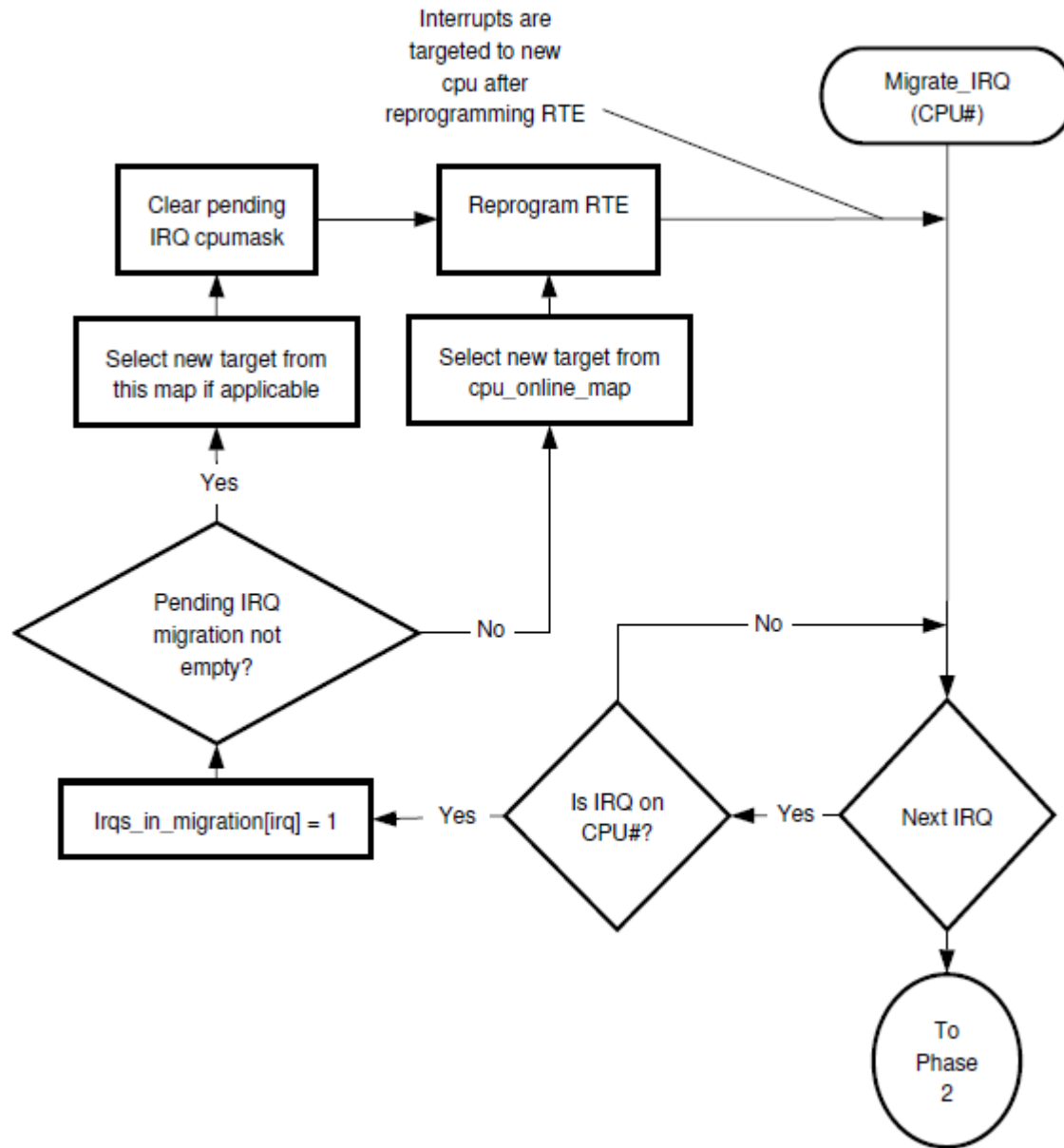
2. MP decision & CPU Hotplug(2)

(*) *mpdecision*에 의해 아래 CPU1이 내려 갈 수도, 다시 올라갈 수도 있음.



```
root@android:/ # cat /proc/interrupts
CPU0          CPU1
16:             0             0      GIC    dg_timer
17:          1285          1469      GIC    gp_timer
19:             0             0      GIC    apps_wdog_bark
48:             21             0      GIC    msgpio
51:            94             0      GIC    rpm_drv
52:             0             0      GIC    mpm_drv
53:             0             0      GIC    rpm_err
54:             0             0      GIC    pm_drv
68:             0             0      GIC    modem_wdog
69:            99             0      GIC    smd_dev
70:             2             0      GIC    smsm_dev
79:             0             0      GIC    vpe
80:             0             0      GIC    vfe
81:             0             0      GIC    vidc
105:            0             0      GIC    msm_rotator
107:            0             0      GIC    MDP
111:            0             0      GIC    hdmi_msm_isr
112:           144             0      GIC    kgs1-3d0
113:             0             0      GIC    kgs1-2d0
114:            11             0      GIC    MIPI_DSI
115:             0             0      GIC    csic
116:             0             0      GIC    csic
119:             0             0      GIC    q6_wdog
121:             0             0      GIC    smsm_dev
122:            24             0      GIC    smd_dev
132:           522             0      GIC    msm_otg, msm_hsus
133:           335             0      GIC    msm-sdcc (cmd), msm-sdcc (pio)
134:             0             0      GIC    msm-sdcc (cmd), msm-sdcc (pio)
135:            13             0      GIC    msm-sdcc (cmd), msm-sdcc (pio)
136:          6812             0      GIC    msm-sdcc (cmd), msm-sdcc (pio)
170:             0             0      GIC    smd_dev
179:            85             0      GIC    qup_err_intr
183:             9             0      GIC    qup_err_intr
185:             3             0      GIC    qup_err_intr
187:            50             0      GIC    qup_err_intr
191:             5             0      GIC    qup_err_intr
193:             0             0      GIC    qup_err_intr
```

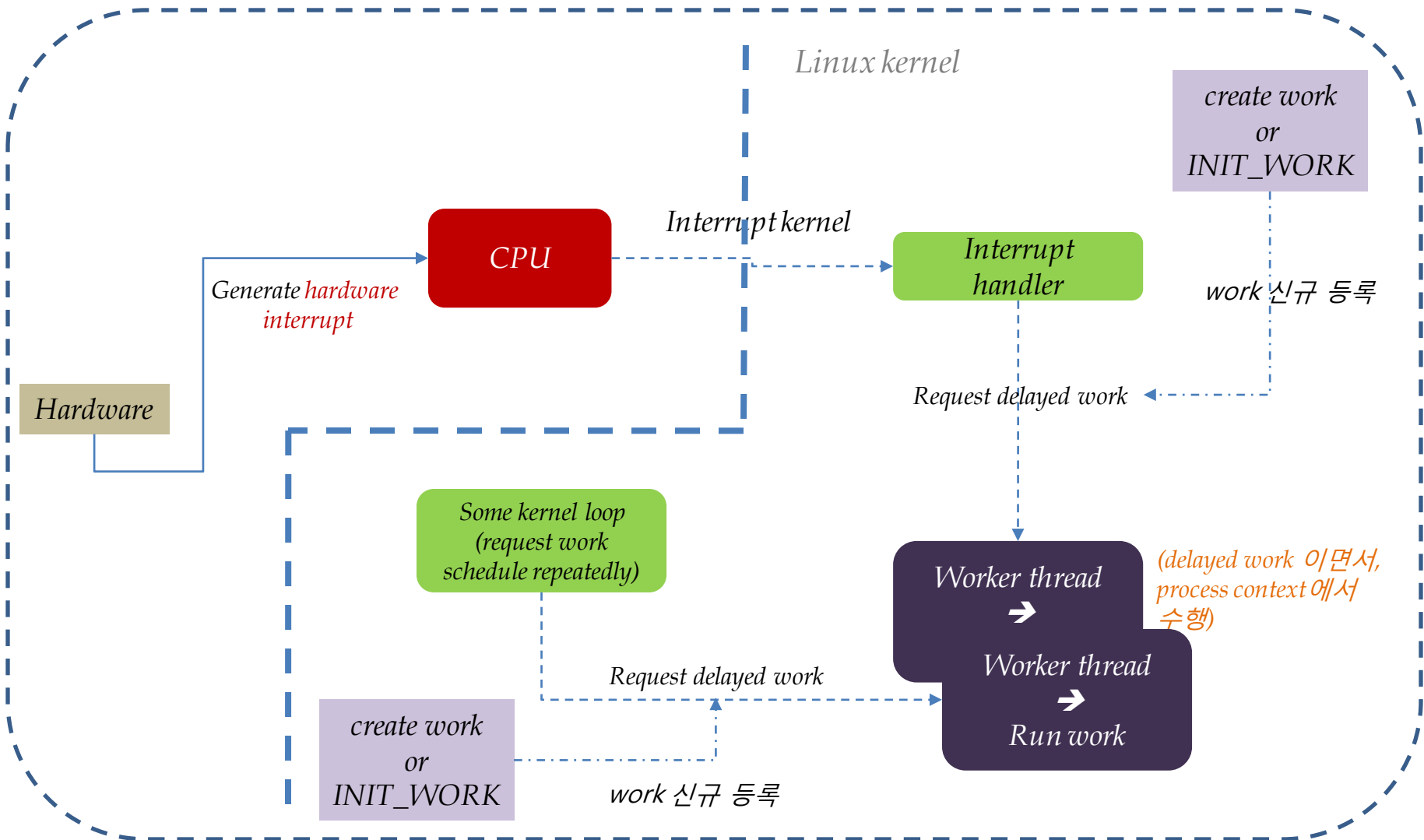

3. IRQ Migration



Work Queue

old workqueue

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



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

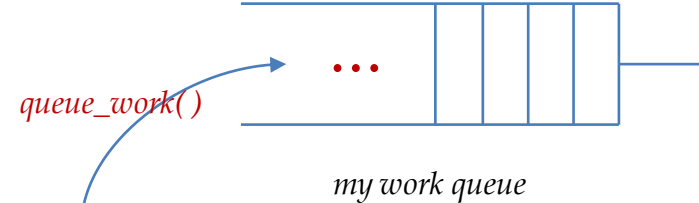
```
struct workqueue_struct {  
    struct cpu_workqueue_struct cpu_wq[NR_CPUS];  
    struct list_head list;  
    const char *name;  
    int singlethread;  
    int freezeable;  
    int rt;  
};
```

```
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>

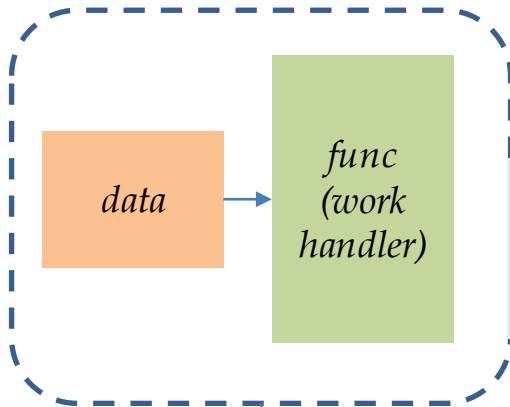


<worker thread flow>

- 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.

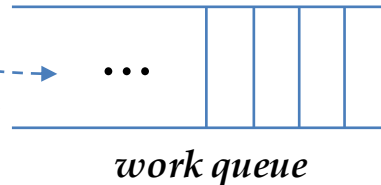
1. Top Half, Bottom Halves and Deferring Work - Work Queue(default)*

<my work - *work_struct*>



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

schedule_work
or
schedule_delayed_work

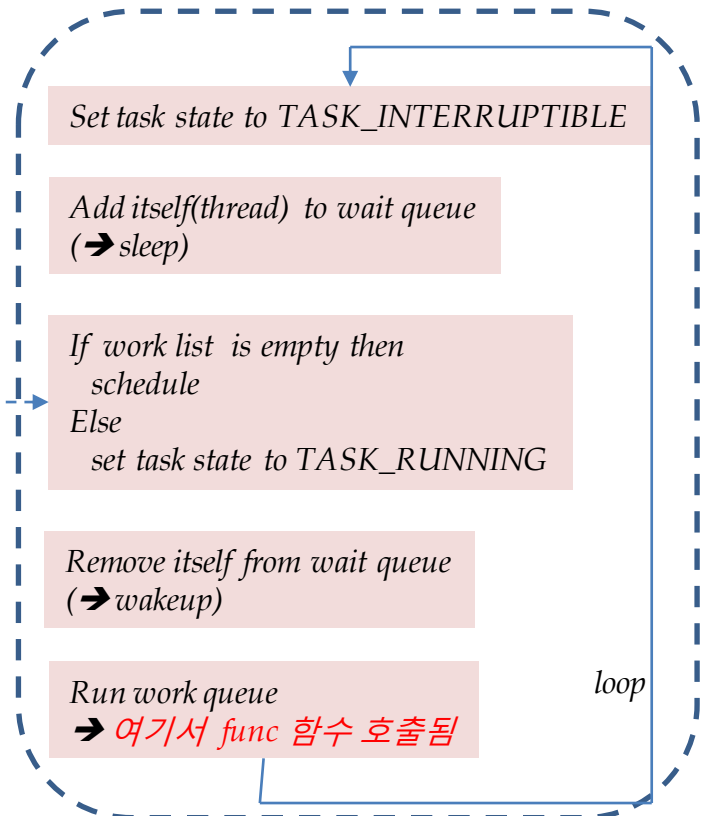


<기 정의된 *work thread*를 사용하는 경우>

```
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.
```

(*) *work queue*라고 하면, *work*, *queue*, *worker thread*의 세가지 요소를 통칭함.
(*) *work queue*를 위해서는 반드시 *worker thread*가 필요함.
(*) 기 정의된 *worker thread(events/0)*를 사용하는 방식과 새로운 *worker thread* 및 *queue*를 만드는 두 가지 방법이 존재함.
(*) *work queue*에서 사용하는 *work*는 *sleep*이 가능하다.
(*) *worker thread* 관련 보다 자세한 사항은 *kernel/workqueue.c* 파일 참조

<*worker thread = events/0*>



1. 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 worker thread를 생성시켜 줌.

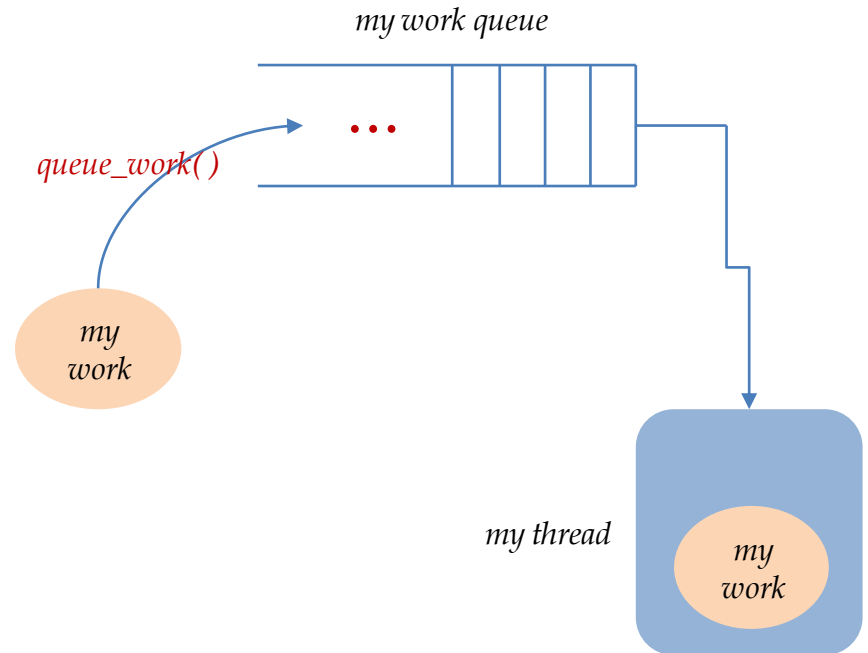
```
int queue_work(struct workqueue_struct *wq, struct  
              work_struct *work);
```

→ 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

1. 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 처  
리하여, queue를 비우도록 요청
```

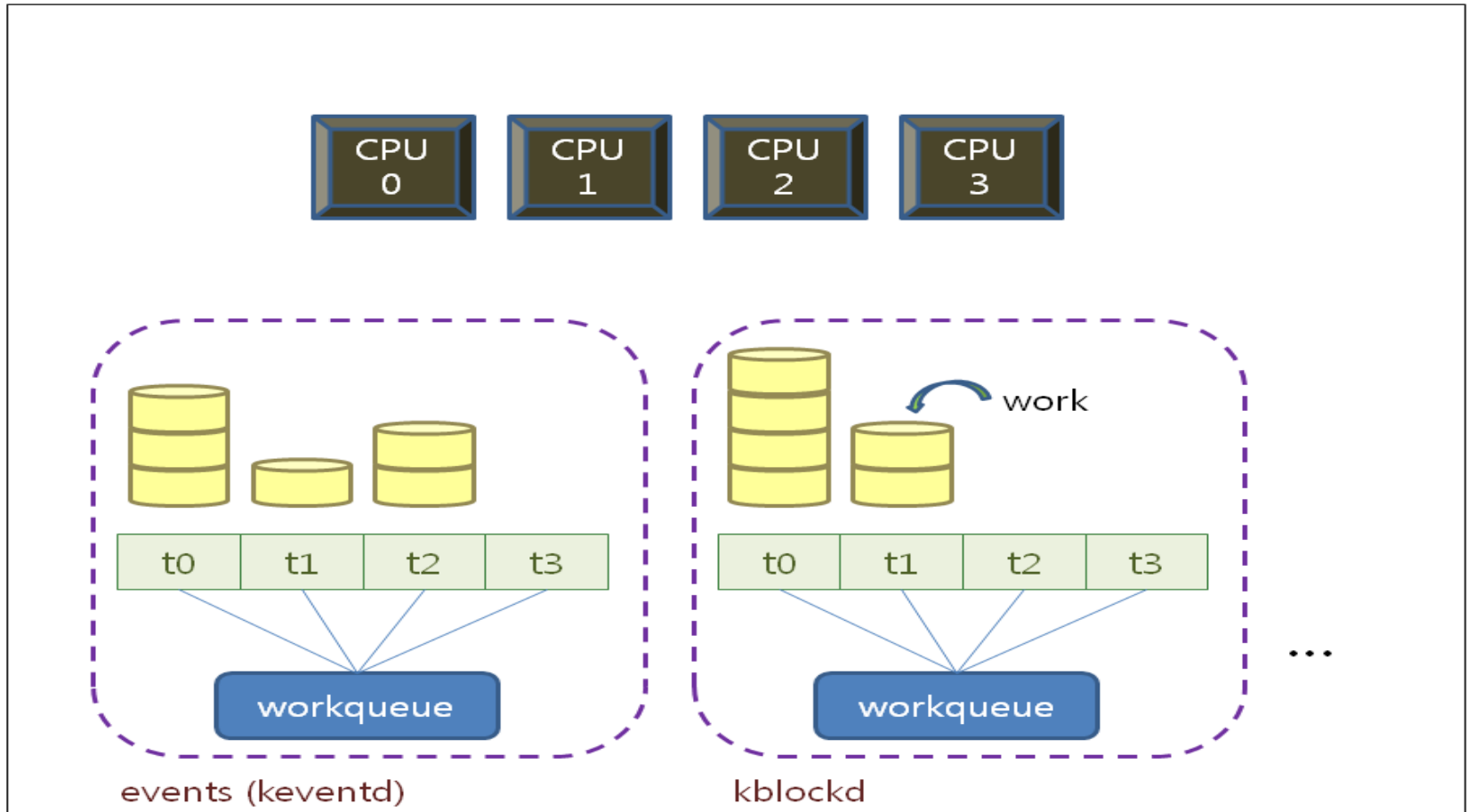
(*) __create_workqueue() 함수의 argument 값에 따라
4가지의 macro가 존재함 !!!
→ 자세한 사항은 workqueue.h 파일 참조

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  
}
```


new workqueue
: *cmwq*(concurrency managed wq)

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

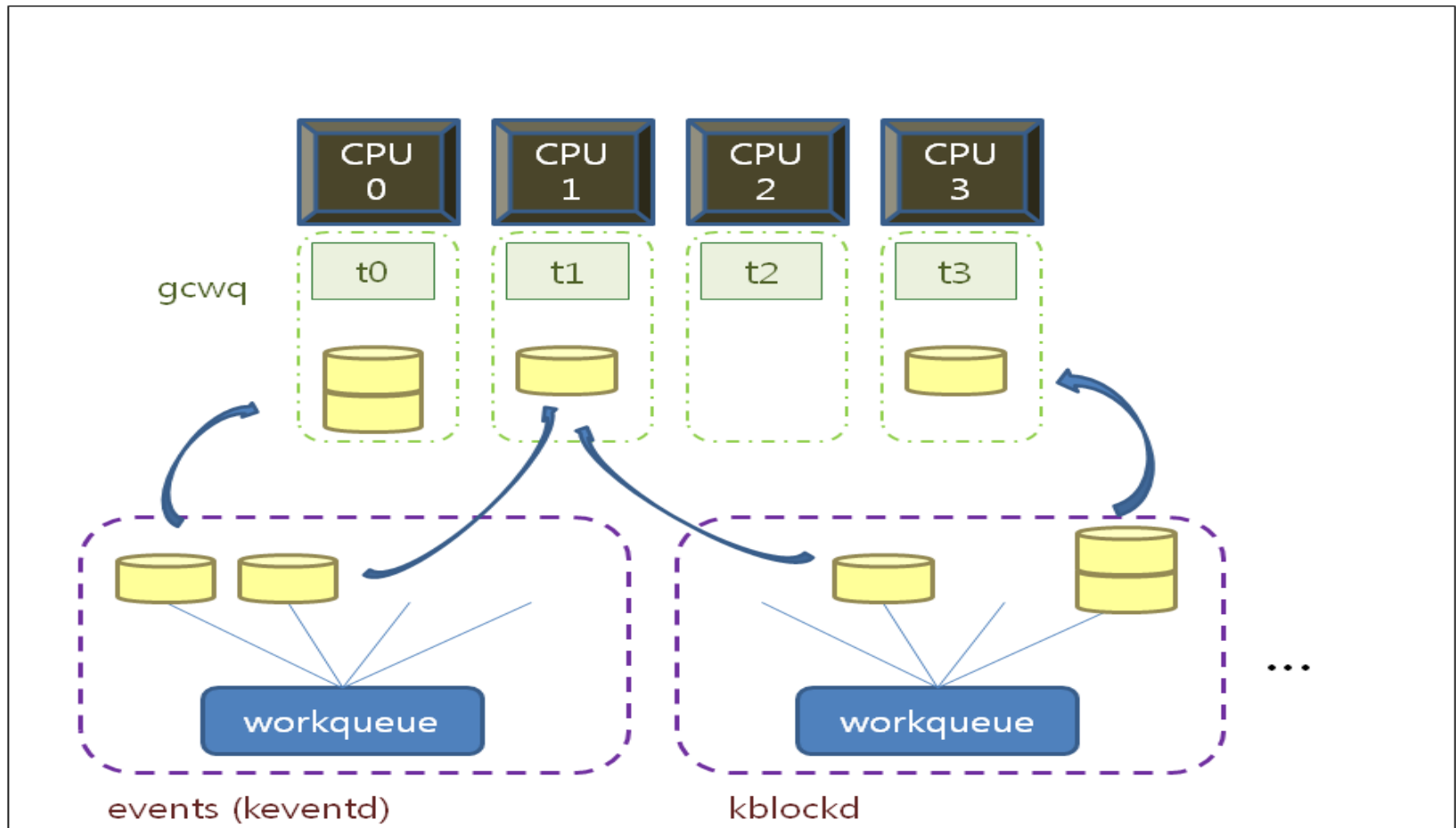


(*) 위의 그림은 2.6.35까지 사용해오던 workqueue의 구조를 표현한 것으로, 아래 site에서 복사해온 것임.

→ <http://studyfoss.egloos.com/5626173>

(*) 주요 특징은 workqueue별로 worker(thread)가 할당되어지며, thread간에는 서로 교류하지 않으므로, 많은 resource(thread - task/pid)를 사용하게 되며, cpu 효율도 떨어지는 방식으로 볼 수 있음.

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



(*) 반면에 새로 구현된 *cmwq*는 CPU별로 *workqueue*(*gcwq*)가 할당되어지며, *thread*간에는 서로 교류하게 되므로, 많은 *resource*(*thread* - *task*/*pid*)를 사용하지 않으며, *cpu* 효율도 높일 수 있는 방식으로 볼 수 있음.

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

cmwq의 설계 목표

1) *Maintain compatibility with the original workqueue API.*

→ 기존 wq API와의 호환성 유지

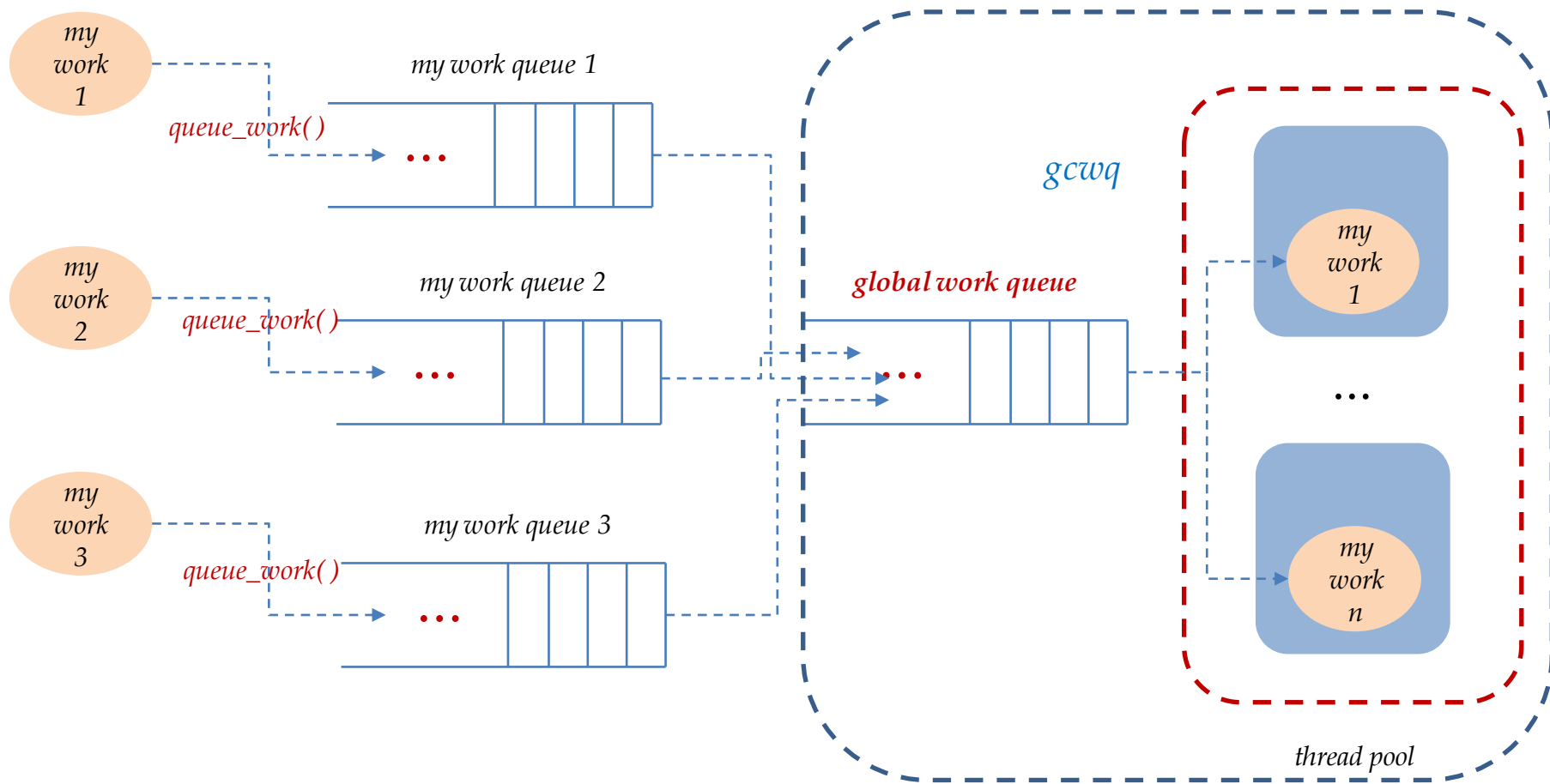
2) *Use per-CPU unified worker pools shared by all wq to provide flexible level of concurrency on demand without wasting a lot of resource.*

→ resource(thread – task/pid) 낭비를 막기 위하여 모든 wq에 의해 제공된 worker(thread) pool을 공유 가능하도록 만듦(단, 동일 CPU 내에서만)

3) *Automatically regulate worker pool and level of concurrency so that the API users don't need to worry about such details.*

→ 사용자가 세세하게 신경 쓰지 않아도 되도록 자동으로 알아서 worker pool과 concurrency level을 조정해 줌.

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



- (*) workqueue 당 worker thread를 할당하는 것이 아니라, cpu 별로 worker thread를 둬.
- (*) 현재 work이 sleep으로 들어 갈 경우, 그냥 대기하는 것이 아니라, 다른 worker thread를 생성하여 이를 처리
→ concurrency를 높임. 이것이 cmwq의 기본 concept !!!
- (*) 새로운 work이 없고, idle한 thread가 5분간 지속되면, 해당 thread를 제거함.
→ resource를 효율적으로 관리함. 보통은 CPU마다 2개 정도의 thread가 존재한다고 보면 됨.

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

```
26 0 0 DW [kworker/u:1]
5518 0 0 SW [kworker/0:2]
9407 0 0 SW [kworker/u:2]
10673 0 0 SW [kworker/u:0]
12015 0 0 SW [kworker/0:0]
12040 0 0 SW [kworker/0:1]
12045 0 0 SW [kworker/u:3]
```

- (*) 위의 그림은 *gcwq(global per-cpu workqueue)*에서 생성하는 *worker thread*를 화면 *capture*한 것임.
- (*) *kworker/0:0*의 첫번째 숫자는 *cpu*를 나타내고, 두번째 숫자는 *thread id*를 의미함.
- (*) 앞서도 설명한 바와 같이, 기존 방식의 경우 *workqueue* 별로 *thread*를 생성하므로, *thread*가 무한정(?) 늘어나는 형태가 되지만, 새로 도입된 *cmwq*에서는 위의 그림에서와 같이 *kworker thread*가 제한된 범위 내에서 늘었다 줄었다를 반복하게 되므로, 보다 효율적이라고 할 수 있음.

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

<일반적인 workqueue 사용 절차>

[1] struct delayed_work my_work;

→ delayed work 변수 선언

[5] if (delayed_work_pending(&my_work))
cancel_delayed_work_sync(&my_work);

→ 대기중인 delayed work을 취소하는 경우 사용

[4] schedule_delayed_work(&my_work, 10);

→ 지연 시간 후, work 실행 요청 시 사용

→ 이 함수가 호출되면, delay time(여기서 10) 후, work function(여기서는 my_work_func)이 호출됨.

[6] cancel_delayed_work_sync(my_work);

→ 대기중인 delayed work을 취소 요청, driver 종료시 호출

[3] static void my_work_func(struct work_struct *work)

```
{  
    /* ... */  
}
```

→ work function 정의

[2] INIT_DELAYED_WORK(&my_work, my_work_func);

→ work 함수 초기화, driver 시작(init 혹은 probe 단계) 시, 호출

<보충 설명>

[1] delayed work 선언

[2] delayed work의 work function 초기화

[3] work function 기술

→ 이 함수가 worker thread에서 수행됨.

[4] work function을 실행하도록 worker thread에 요청

<요청 시점>

→ interrupt handler 내에서 요청

→ init 함수 내에서 요청

→ 특정 함수 내에서 요청

→ work function 내에서 자신을 다시 호출(recursively)

등 요청하는 시점이 다양함.

[5] 새로운 요청을 하기 전에, 보통 기존에 요청한 내용을 취소한 후, 하고자 할 경우에 사용함.

[6] 드라이버 동작을 종료할 경우, 기존에 요청한 work을 취소하도록 함.

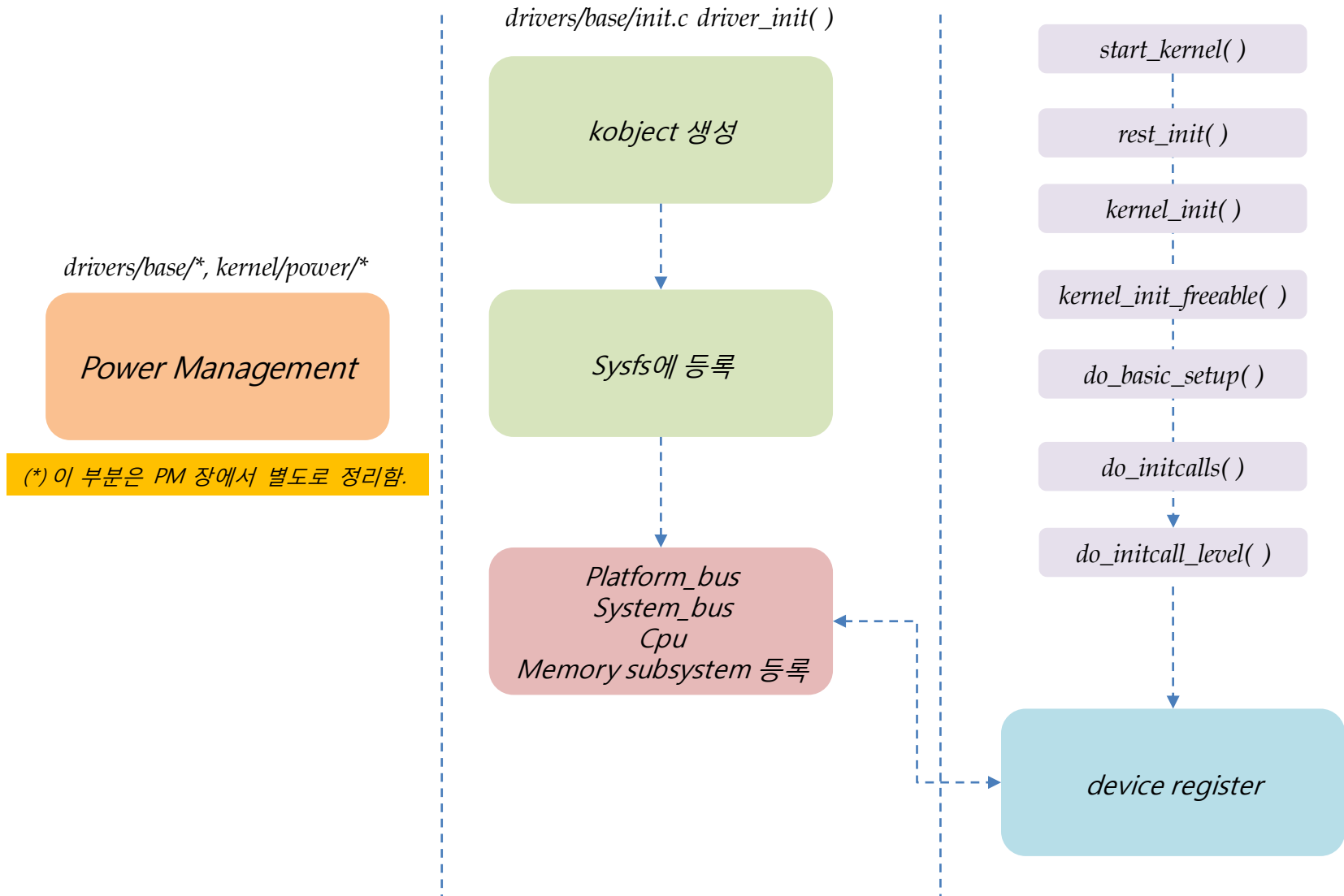
(*) 사용 방식은 기존 방식과 동일함.

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

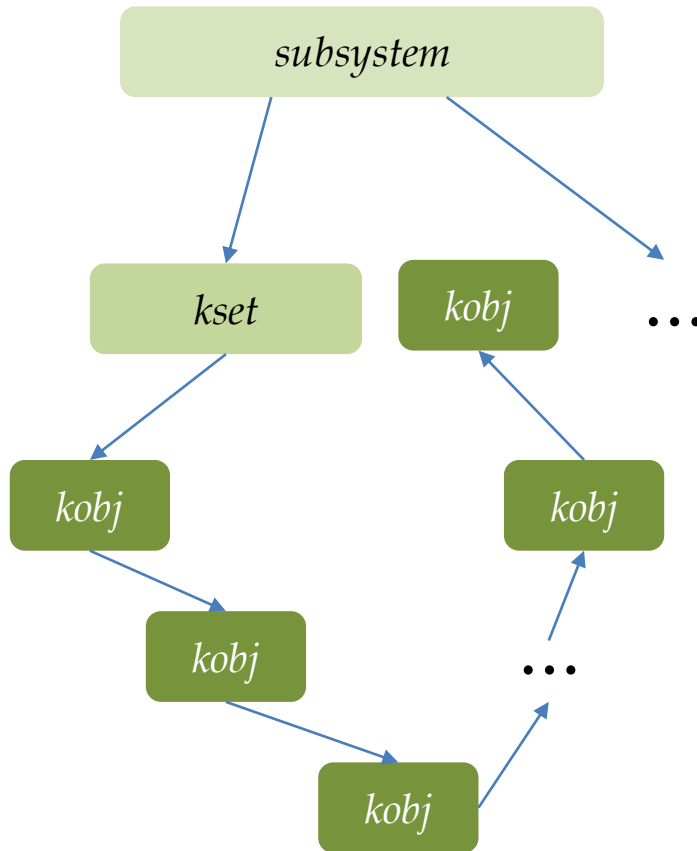
- *cmwq* 관련 보다 자세한 사항은 Documentation/workqueue.txt 파일을 참조
- 혹은 아래 site 참조
- <http://studyfoss.egloos.com/5626173>

Driver Initialization

1. Driver Initialization Overview



2 kobjects & sysfs



```
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 model*을 위해 등장한 것...

➔ *Kset*은 *kobject*의 묶음이고, *subsystem*은 *kset*의 묶음임.

(*) *sysfs*는 *kobject*의 계층 *tree*를 표현(*view*)해 주는 *memory 기반의 file system*으로 2.6에서 부터 소개된 방법 ➔ *kernel device*와 *user process*가 *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 !!!

3. Platform Device & Driver(1) - 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,  
};
```

- **platform_driver 정의 및 초기화**
- **probe/remove**

(drivers/XXXX/xxxx.c 등에 위치함)

.name 필드("bluesleep")로 상호 연결(binding)

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

(*) drivers/base/platform.c

(*) include/linux/platform_device.h 참조

(*) Documentation/driver-model/platform.txt 참조

3. Platform Device & Driver(2) – *platform device data structure*

```
struct platform_device {  
    const char * name;  
    int id;  
    struct device dev;  
    u32 num_resources;  
    struct resource * resource;  
  
    const struct platform_device_id *id_entry;  
  
    /* arch specific additions */  
    struct pdev_archdata archdata;  
};
```

다음 page 참조

```
struct resource {  
    resource_size_t start;  
    resource_size_t end;  
    const char *name;  
    unsigned long flags;  
    struct resource *parent, *sibling,  
    *child;  
};
```

```
struct platform_device_id {  
    char  
    name[PLATFORM_NAME_SIZE];  
    kernel_ulong_t driver_data  
  
    __attribute__((aligned(sizeof(kernel_ulong_t))));  
};
```

(*) 디바이스는 고유의 명칭(id)있는데, platform device의 경우는 platform_device.dev.bus_id가 device를 구분하는 값(canonical name)임.

(*) 이는 platform_device.name과 platform_device.id로 만들어지게 됨.

```

struct device {
    struct device    *parent;

    struct device_private  *p;

    struct kobject kobj;
    const char    *init_name; /* initial name of the device */
    struct device_type  *type;

    struct mutex    mutex; /* mutex to synchronize calls to
                            * its driver.
                            */

    struct bus_type *bus;    /* type of bus device is on */
    struct device_driver *driver; /* which driver has allocated
this
                                device */
    void    *platform_data; /* Platform specific data, device
                                core doesn't touch it */
    struct dev_pm_info  power;

#ifdef CONFIG_NUMA
    int    numa_node; /* NUMA node this device is close to */
#endif
    u64    *dma_mask; /* dma mask (if dma'able device) */
    u64    coherent_dma_mask; /* Like dma_mask, but for
                                alloc_coherent mappings as
                                not all hardware supports
                                64 bit addresses for consistent
                                allocations such descriptors. */

    struct device_dma_parameters  *dma_parms;

```

```

    struct list_head    dma_pools; /* dma pools (if dma'ble)
    */

    struct dma_coherent_mem  *dma_mem; /* internal for
coherent mem
                                override */
    /* arch specific additions */
    struct dev_archdata  archdata;
#ifdef CONFIG_OF
    struct device_node  *of_node;
#endif

    dev_t    devt; /* dev_t, creates the sysfs "dev" */

    spinlock_t    devres_lock;
    struct list_head  devres_head;

    struct klist_node  knode_class;
    struct class  *class;
    const struct attribute_group  **groups; /* optional
groups */

    void (*release)(struct device *dev);
};

```

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

```
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 (*resume)(struct platform_device *);  
    struct device_driver driver; -----  
    const struct platform_device_id *id_table;  
};
```

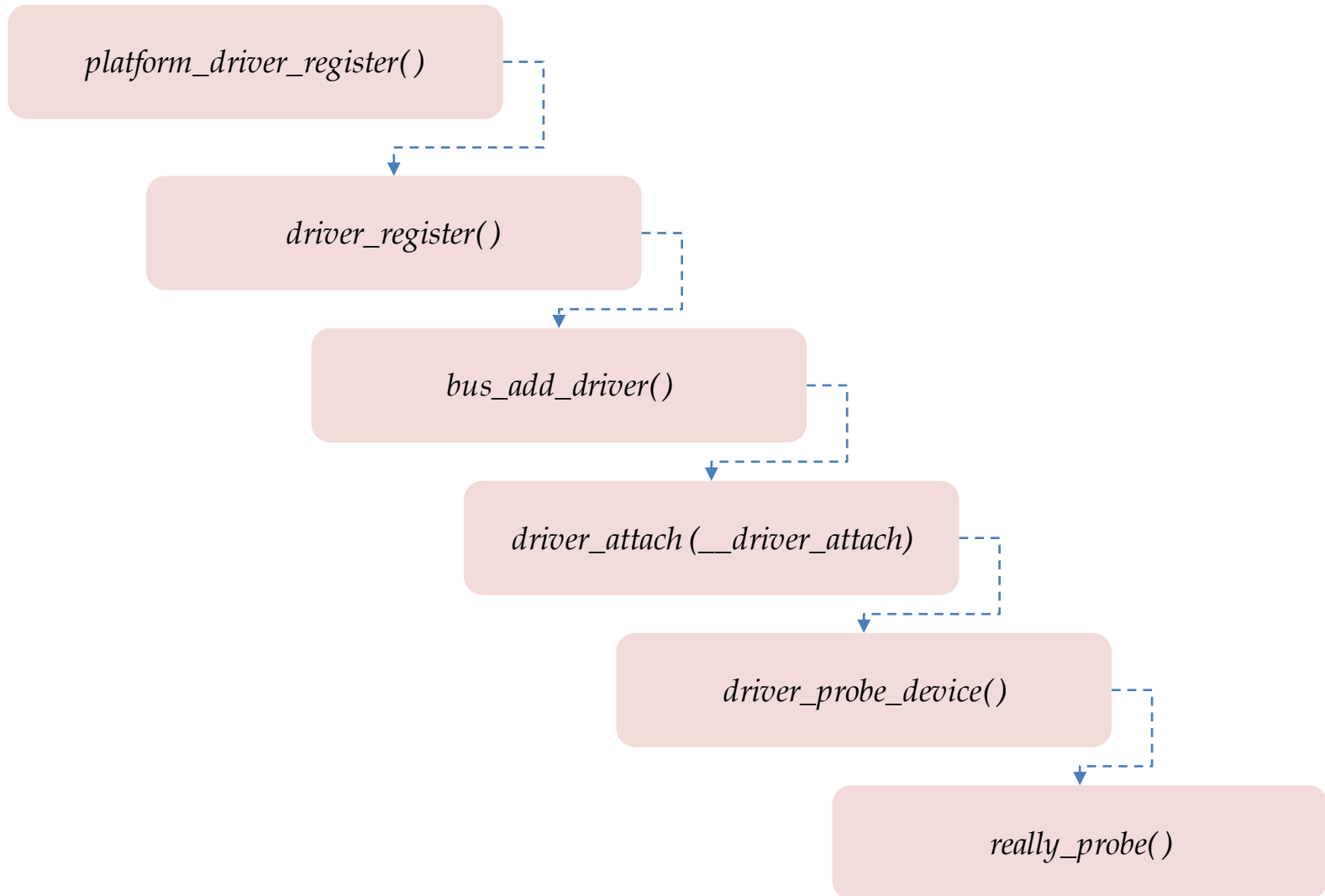
```
struct device_driver {  
    const char    *name;  
    struct bus_type *bus;  
  
    struct module    *owner;  
    const char    *mod_name; /* used for built-in modules */  
  
    bool suppress_bind_attrs; /* disables bind/unbind via sysfs */  
    /*  
#if defined(CONFIG_OF)  
    const struct of_device_id *of_match_table;  
#endif  
  
    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);  
    const struct attribute_group **groups;  
  
    const struct dev_pm_ops *pm;  
  
    struct driver_private *p;  
};
```

```
struct dev_pm_ops {  
    int (*prepare)(struct device *dev);  
    void (*complete)(struct device *dev);  
    int (*suspend)(struct device *dev);  
    int (*resume)(struct device *dev);  
    int (*freeze)(struct device *dev);  
    int (*thaw)(struct device *dev);  
    int (*poweroff)(struct device *dev);  
    int (*restore)(struct device *dev);  
    int (*suspend_noirq)(struct device *dev);  
    int (*resume_noirq)(struct device *dev);  
    int (*freeze_noirq)(struct device *dev);  
    int (*thaw_noirq)(struct device *dev);  
    int (*poweroff_noirq)(struct device *dev);  
    int (*restore_noirq)(struct device *dev);  
    int (*runtime_suspend)(struct device *dev);  
    int (*runtime_resume)(struct device *dev);  
    int (*runtime_idle)(struct device *dev);  
};
```

(*) *platform driver*와 *power management*간의 관계를 보기 위해 정리한 것임.

(*) *suspend/resume callback* 함수가 세군데나 있음.
드라이버 초기화시, 실제로 할당된 *callback*만이 사용될 것임^^

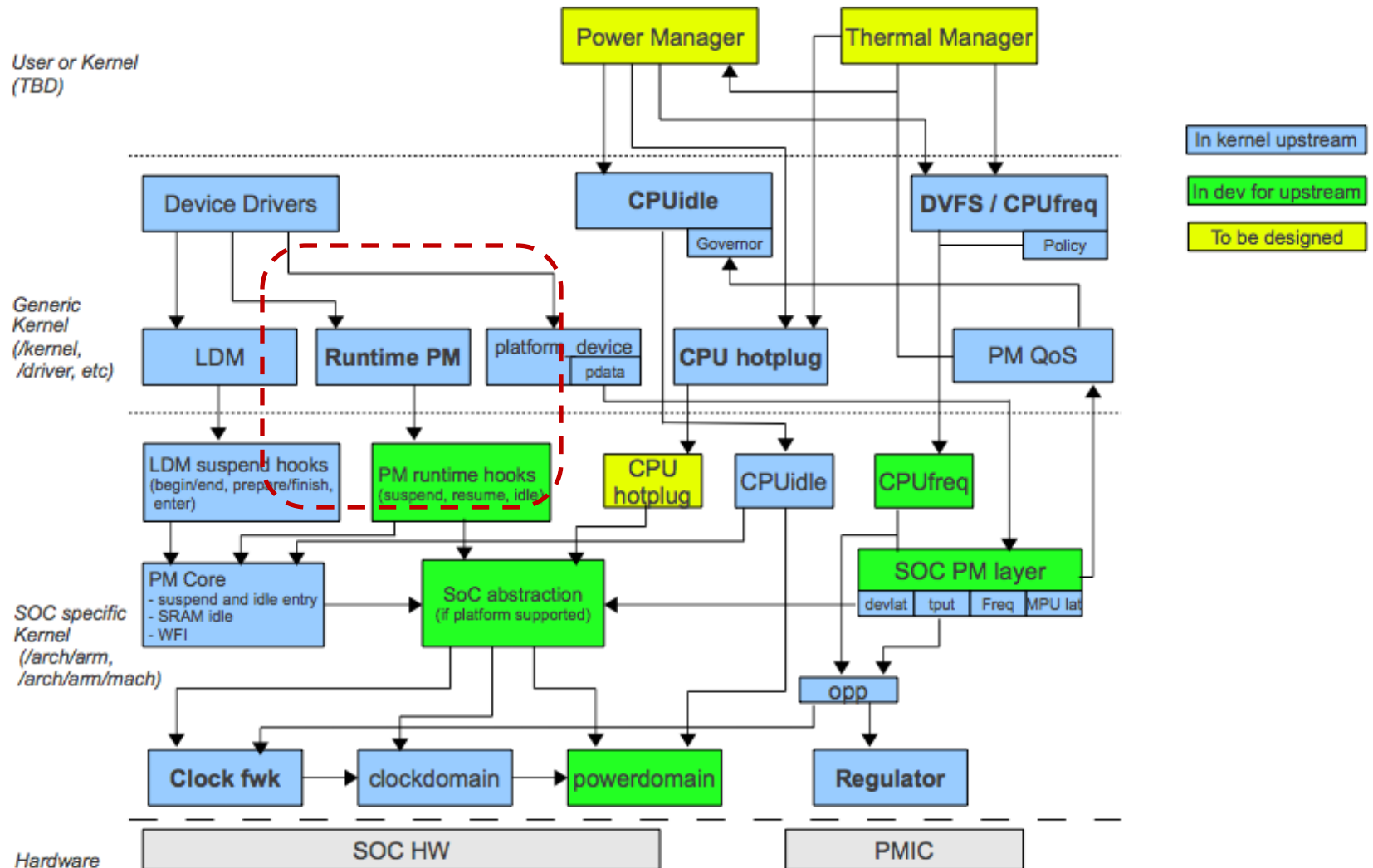
3. Platform Device & Driver(4) - *Probe* 함수 호출 순서



Power Management

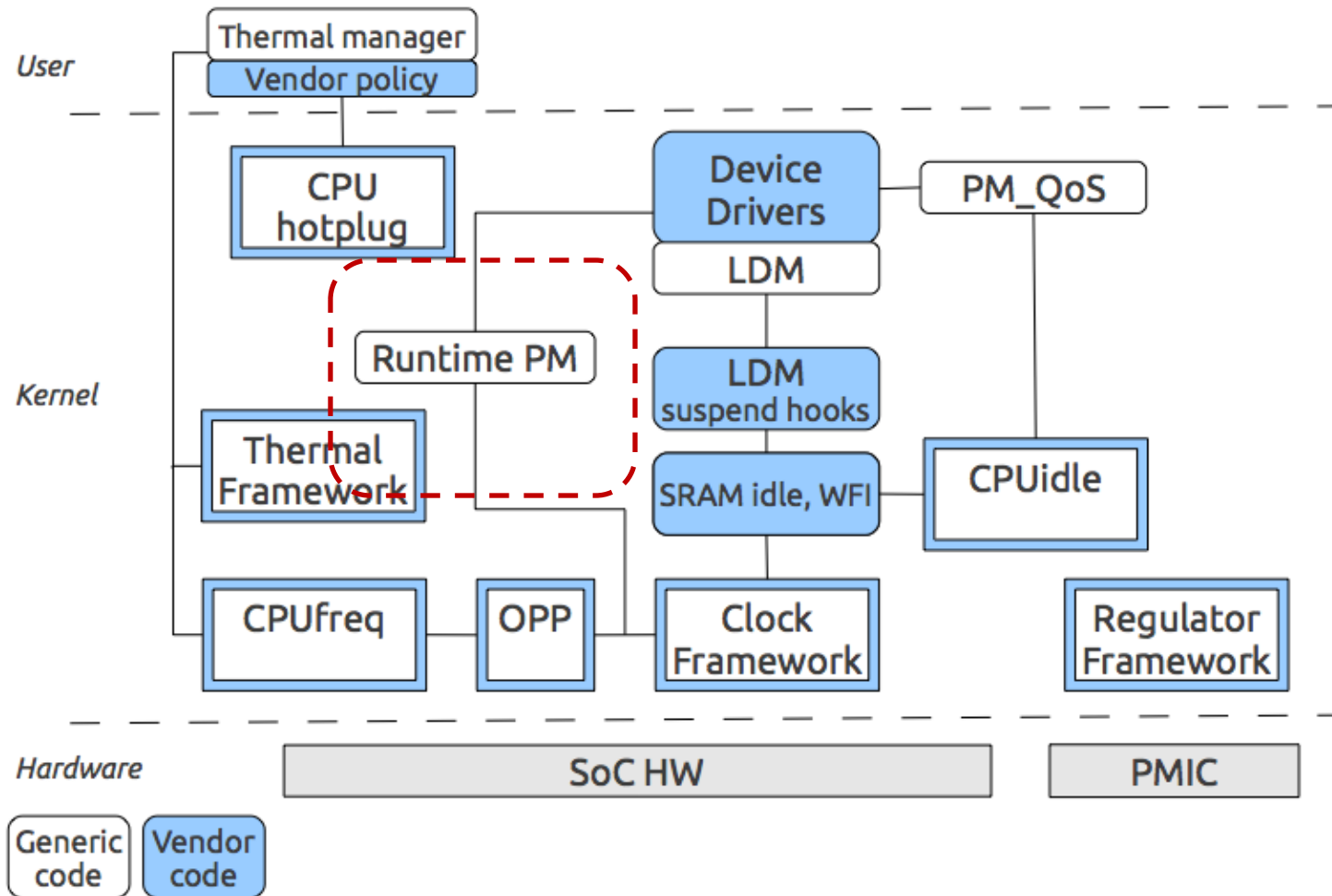
1. Linux PM Frameworks(1)

(*) 아래 그림에는 앞으로 설명할 autosleep & wakeup source 개념은 포함되어 있지 않음.

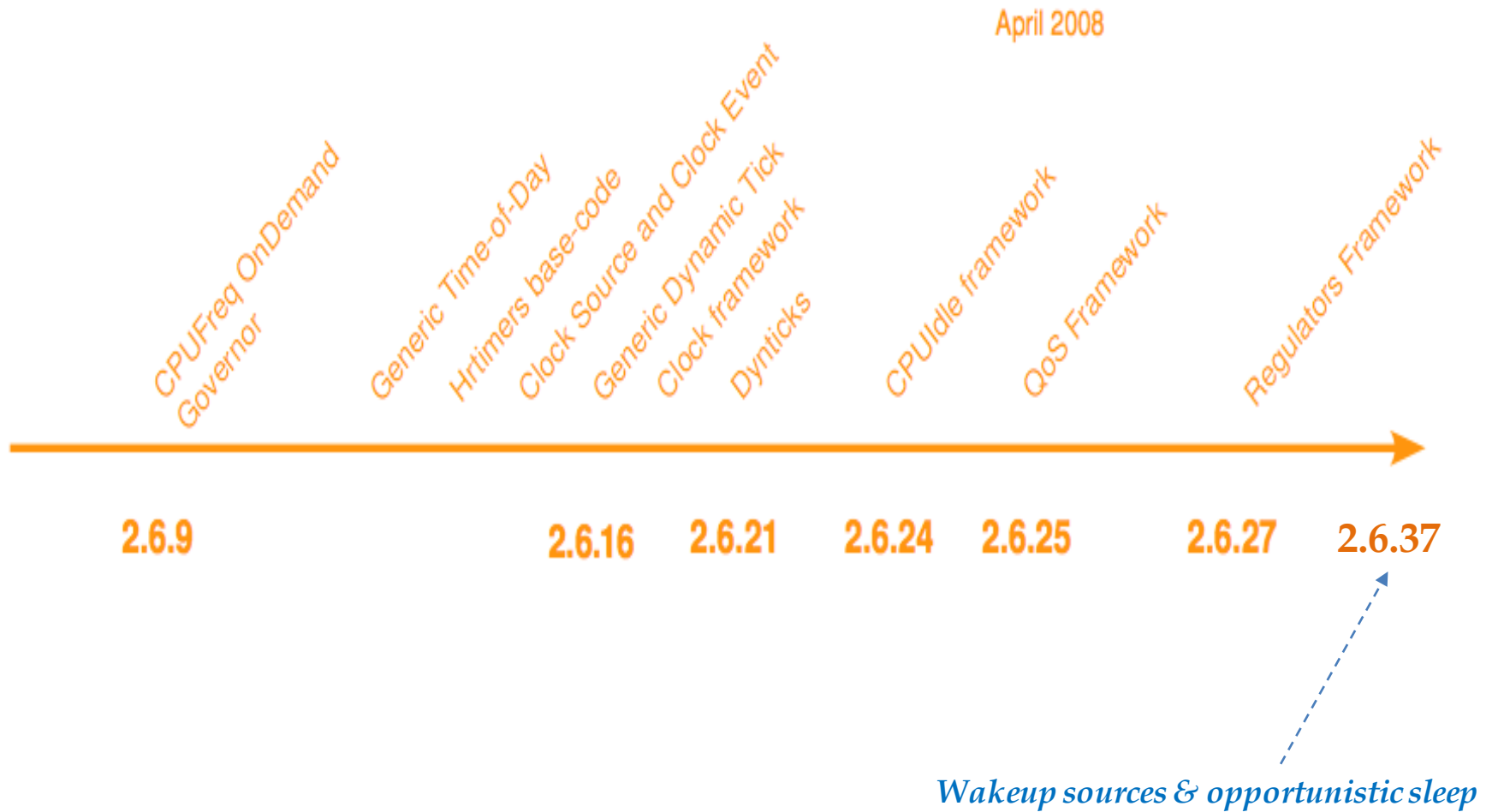


1. Linux PM Frameworks(2)

(*) 아래 그림에는 앞으로 설명할 autosleep & wakeup source 개념은 포함되어 있지 않음.



1. Linux PM Frameworks(3)

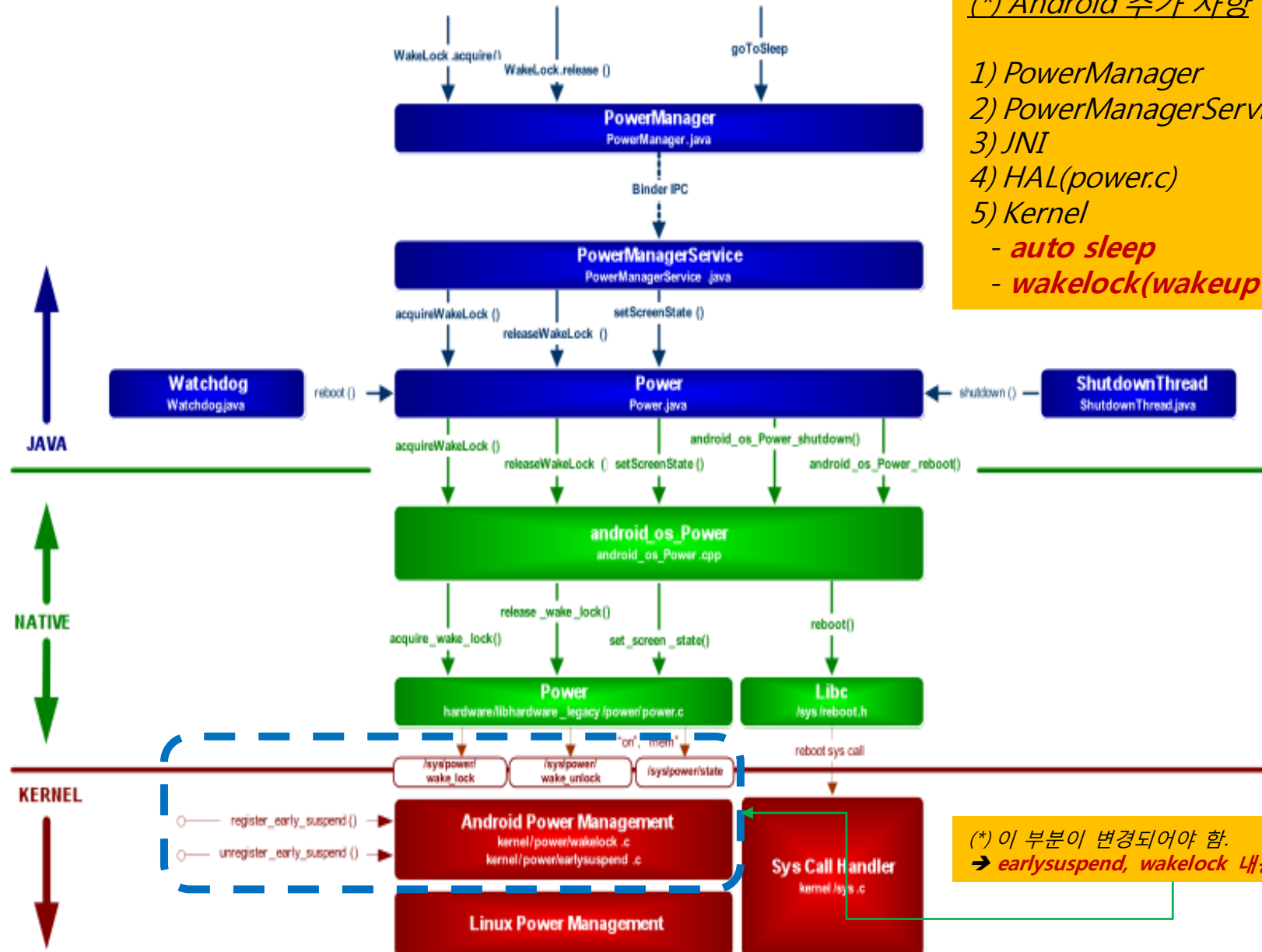


2. Android Power Management - Overview

(*) Android 추가 사항

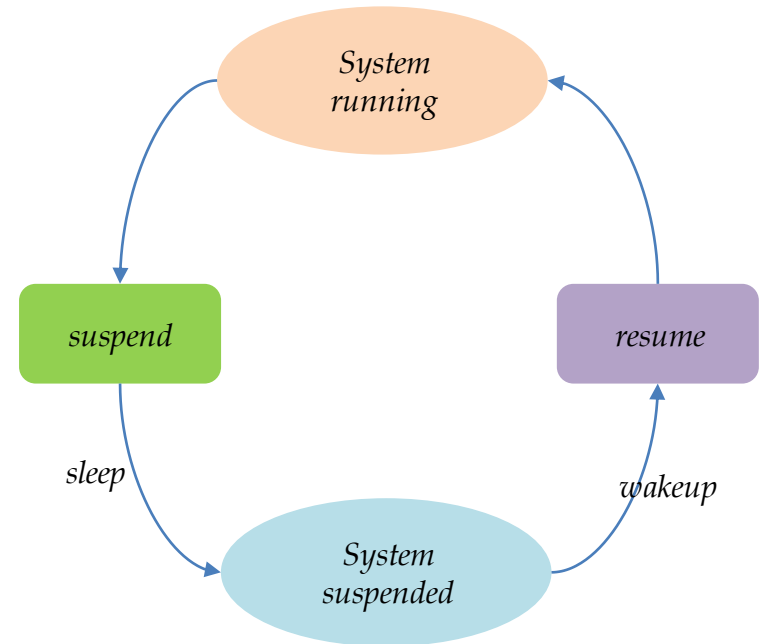
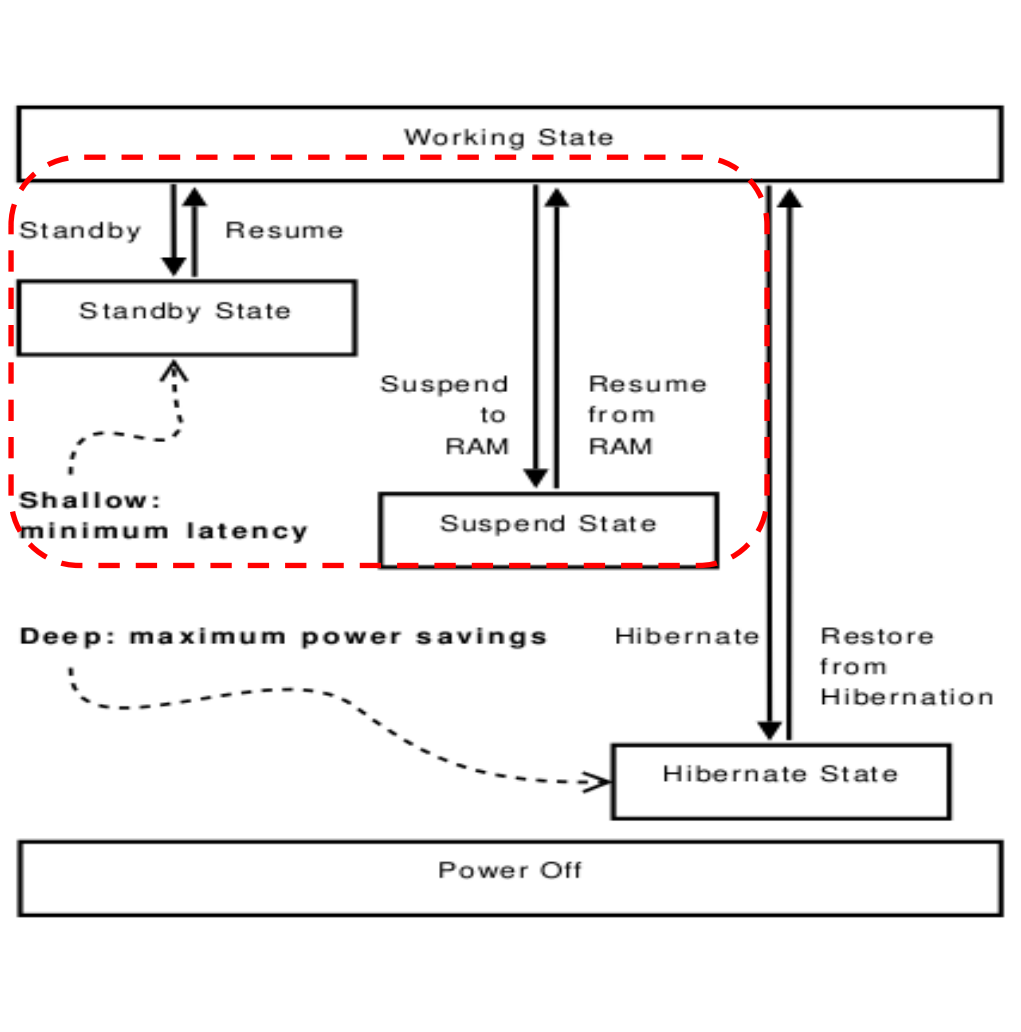
- 1) PowerManager
- 2) PowerManagerService
- 3) JNI
- 4) HAL(power.c)
- 5) Kernel

- **auto sleep**
- **wakelock(wakeup source)**



(*) 이 부분이 변경되어야 함.
→ **earlysuspend**, **wakelock** 내용이 사라짐.

3. PM Old Mechanism(1)



3. PM Old Mechanism(2) – Suspend/Resume

Suspend 절차:

*) *application*으로 부터 “echo mem > /sys/power/state” 형태로 suspend 시작

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

Resume 절차:

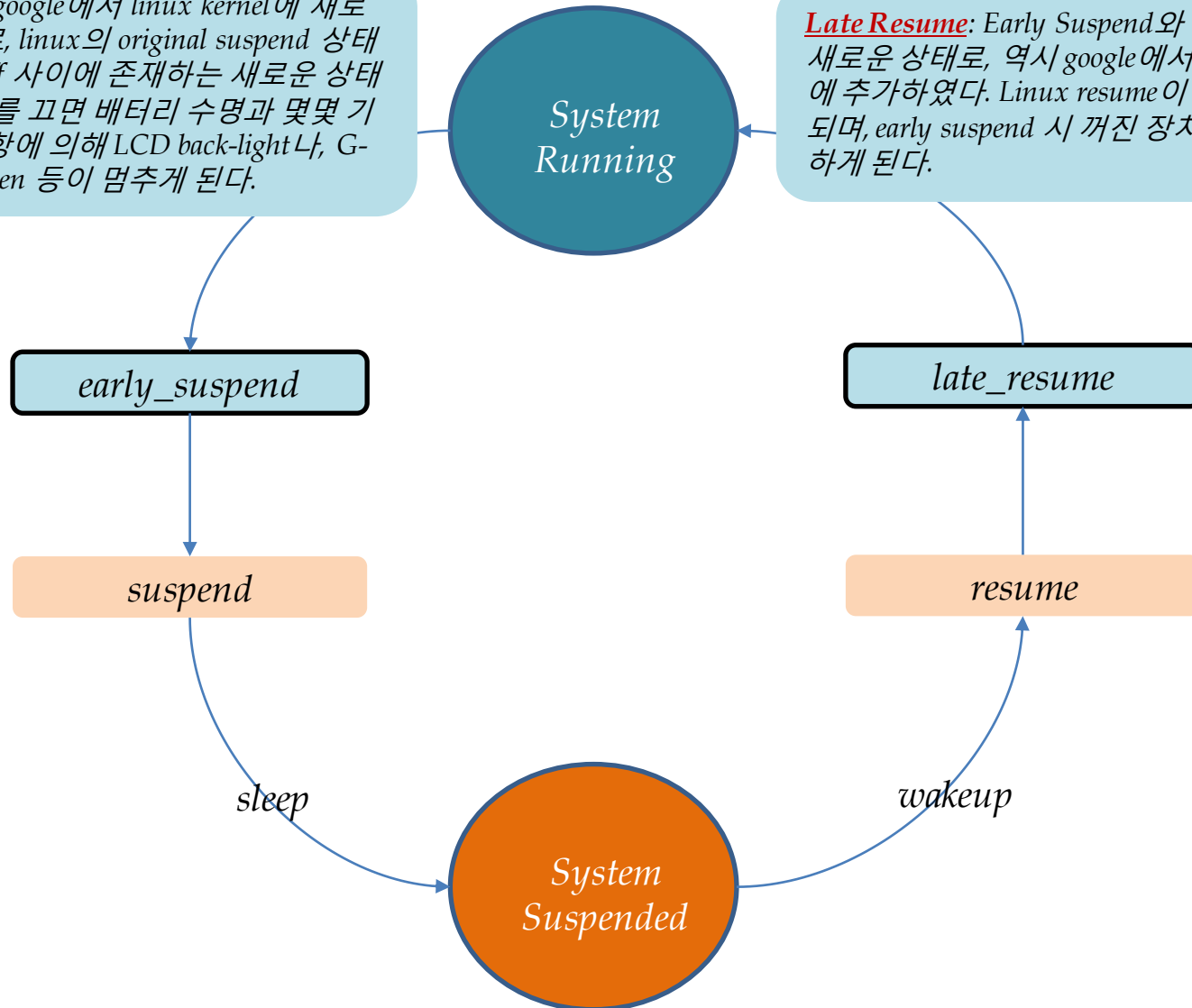
*) *interrupt* 등에 의해 시작됨(예: power key 누름, 전화 걸려옴 ...).
따라서 이와 관련된 장치는 항상 wakeup 상태로 있어야 함.

- 1) System 장치(/sys/devices/system)를 먼저 깨우고,
- 2) IRQ 활성화, CPU 활성화
- 3) 나머지 모든 장치를 깨우고(resume callback 함수 호출), freezing 되어 있는 프로세스와 task를 깨움.

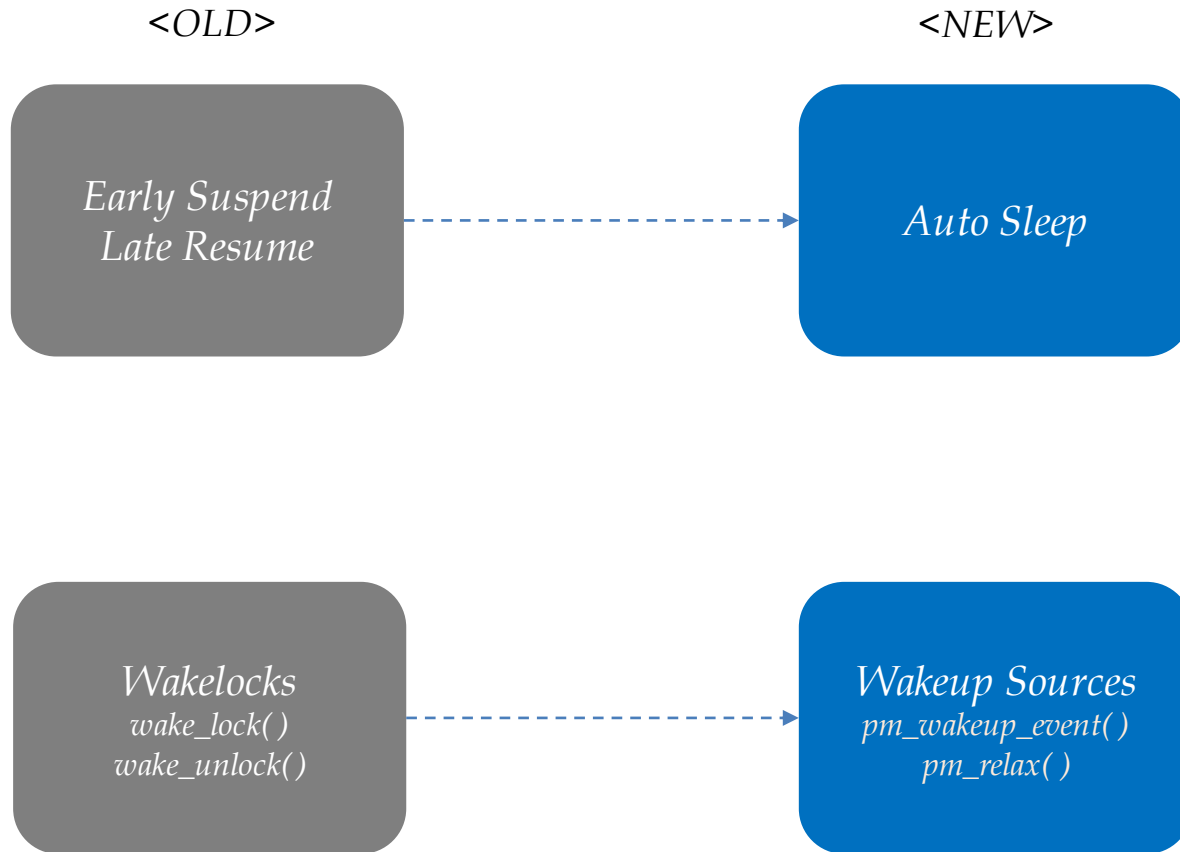
3. PM Old Mechanism(3) – *Early Suspend/Late Resume*

Early Suspend: google에서 linux kernel에 새로 추가한 부분으로, 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.

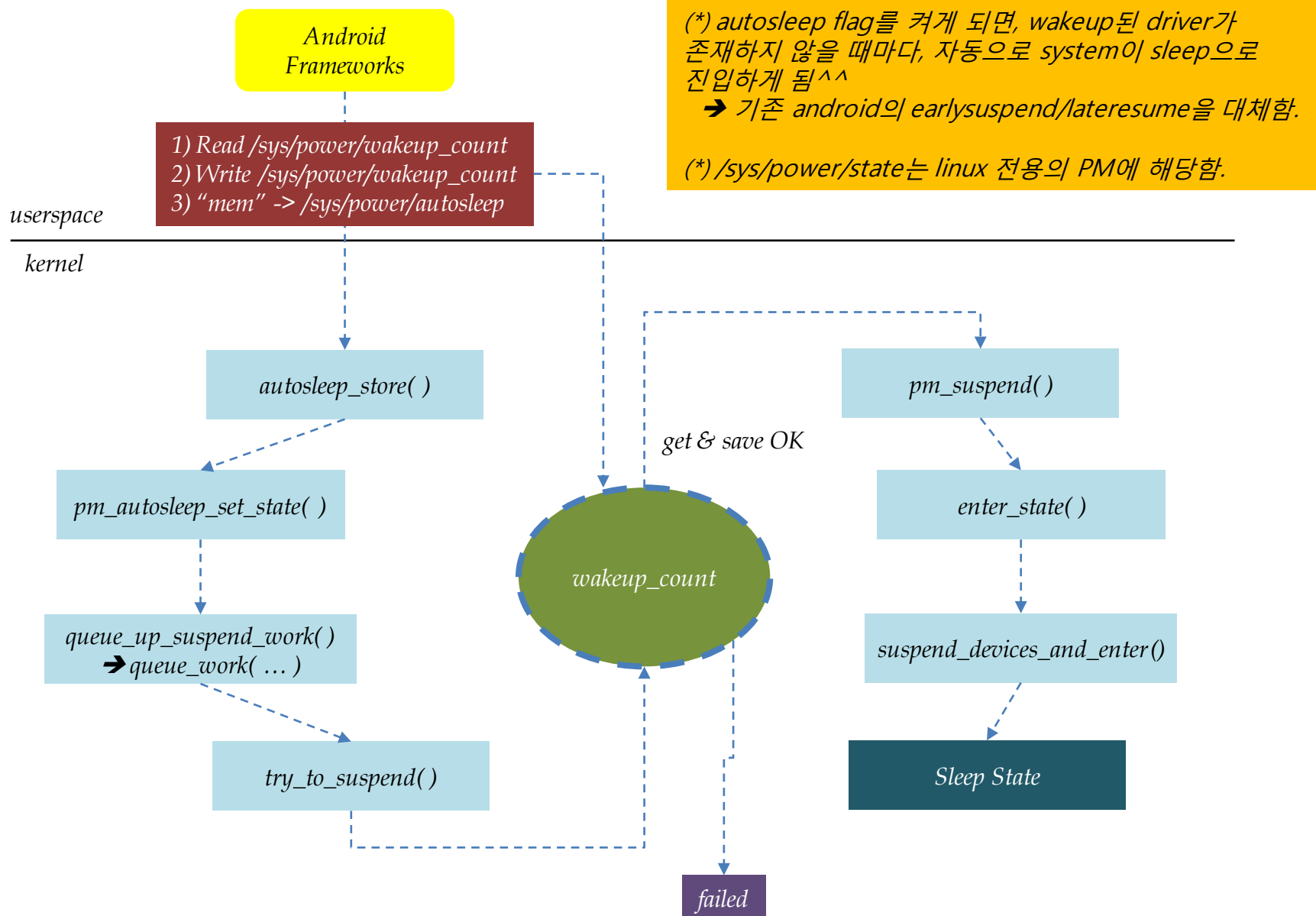


(*) 다음 page로 넘어가기 전에 ...



(*) 단, android와의 호환성을 위해서, userspace 용 wakelock API는 그대로 존재함.
→ 단, wakelock API 내부는 wakeup source로 구현되어 있음.

4. PM New Mechanism(1) – System Sleep Sequence/1



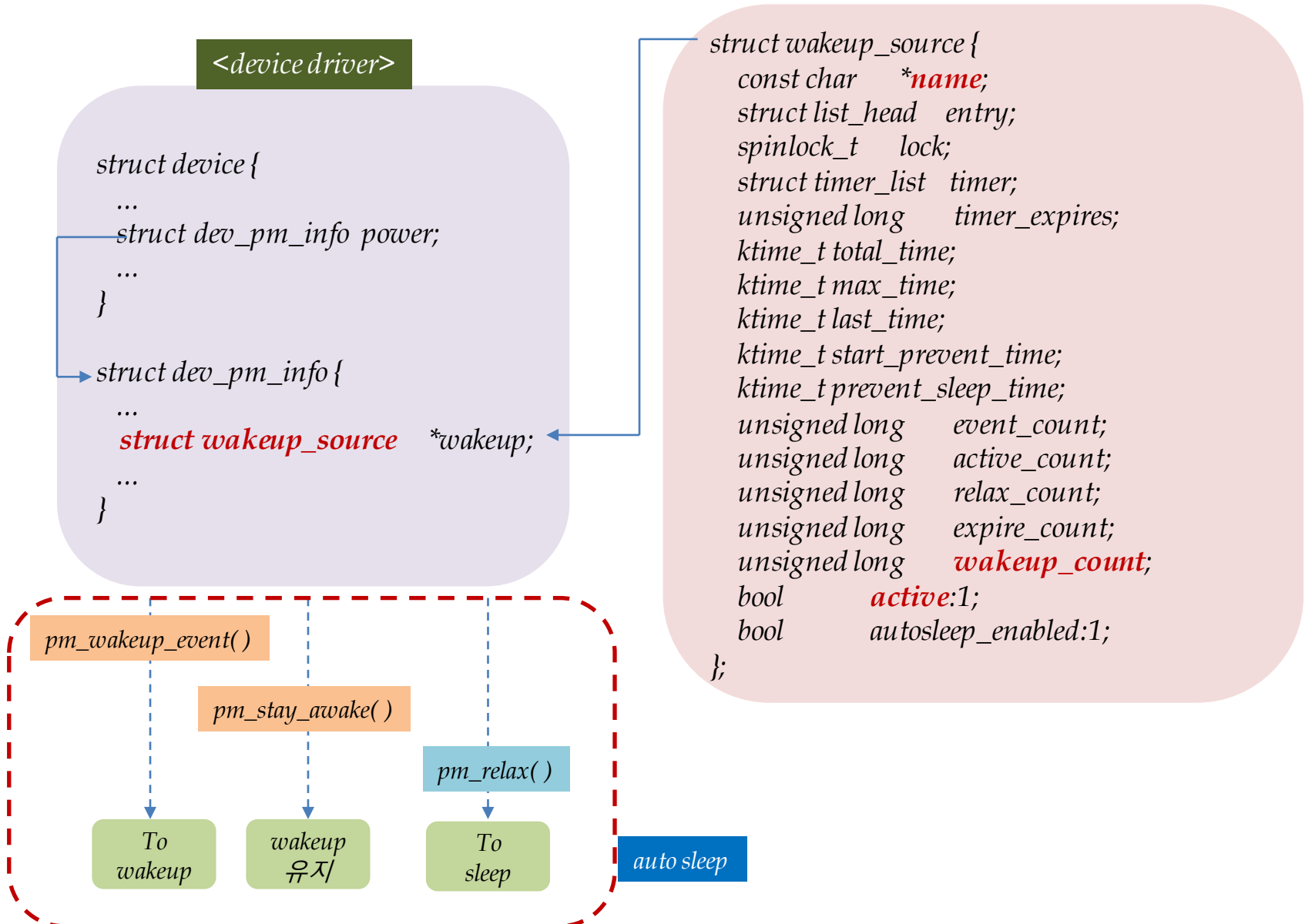
4. PM New Mechanism(1) – System Sleep Sequence/2

<suspend flow>

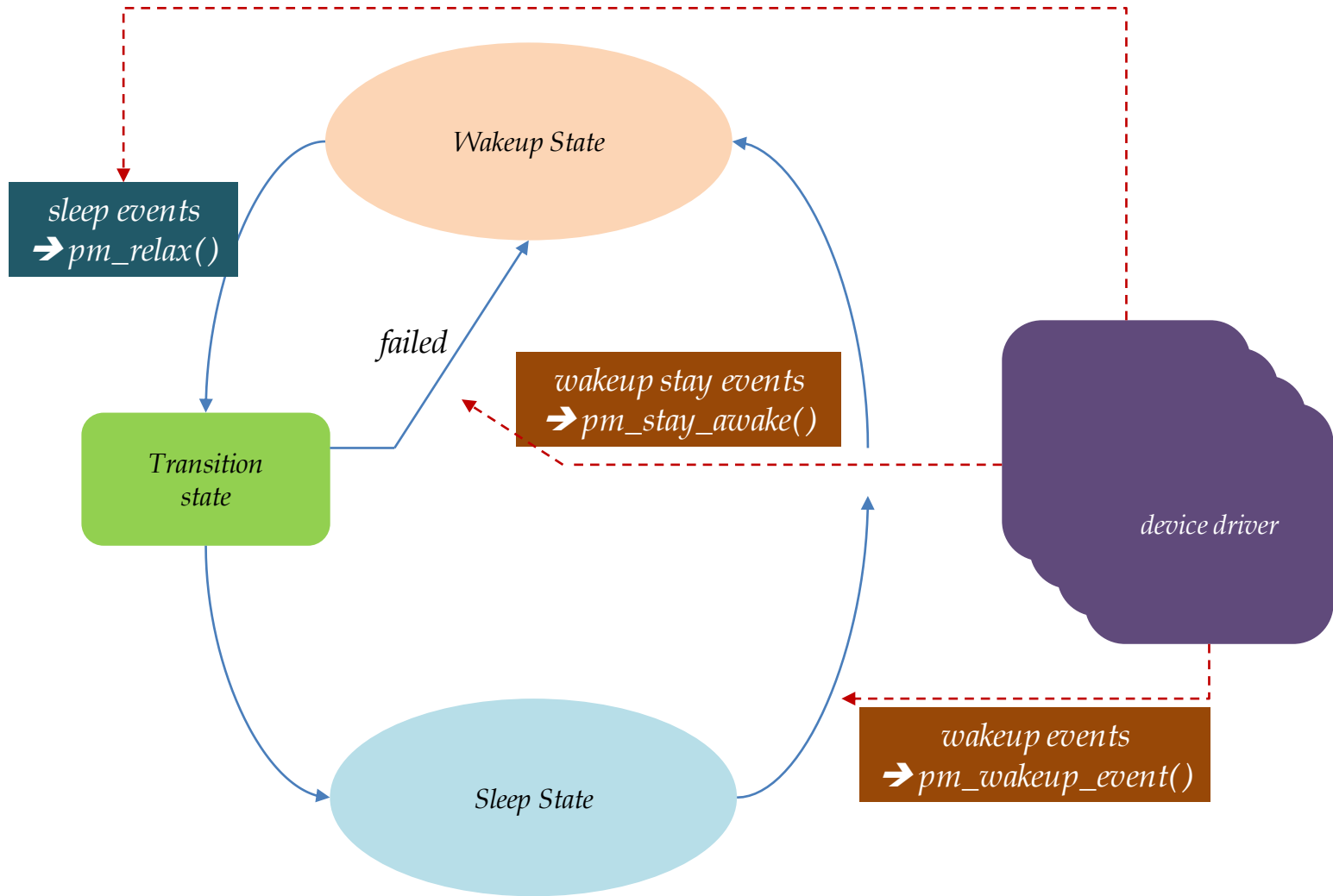
➔ echo "mem" > /sys/power/autosleep
(active wakeup source가 없을 때 마다 suspend로 진입 !!!)

- 1) autosleep_store() kernel/power/autosleep.c
- 2) pm_autosleep_set_state(state) kernel/power/autosleep.c
- 3) queue_up_suspend_work() kernel/power/autosleep.c
 - > queue_work(autosleep_wq, &suspend_work);
 - > DECLARE_WORK(suspend_work, try_to_suspend)
- 4) try_to_suspend() kernel/power/autosleep.c
 - > pm_get_wakeup_count()
 - > pm_save_wakeup_count()
- 5) pm_suspend(autosleep_state) kernel/power/suspend.c
- 6) enter_state() kernel/power/suspend.c
- 7) suspend_devices_and_enter() kernel/power/suspend.c

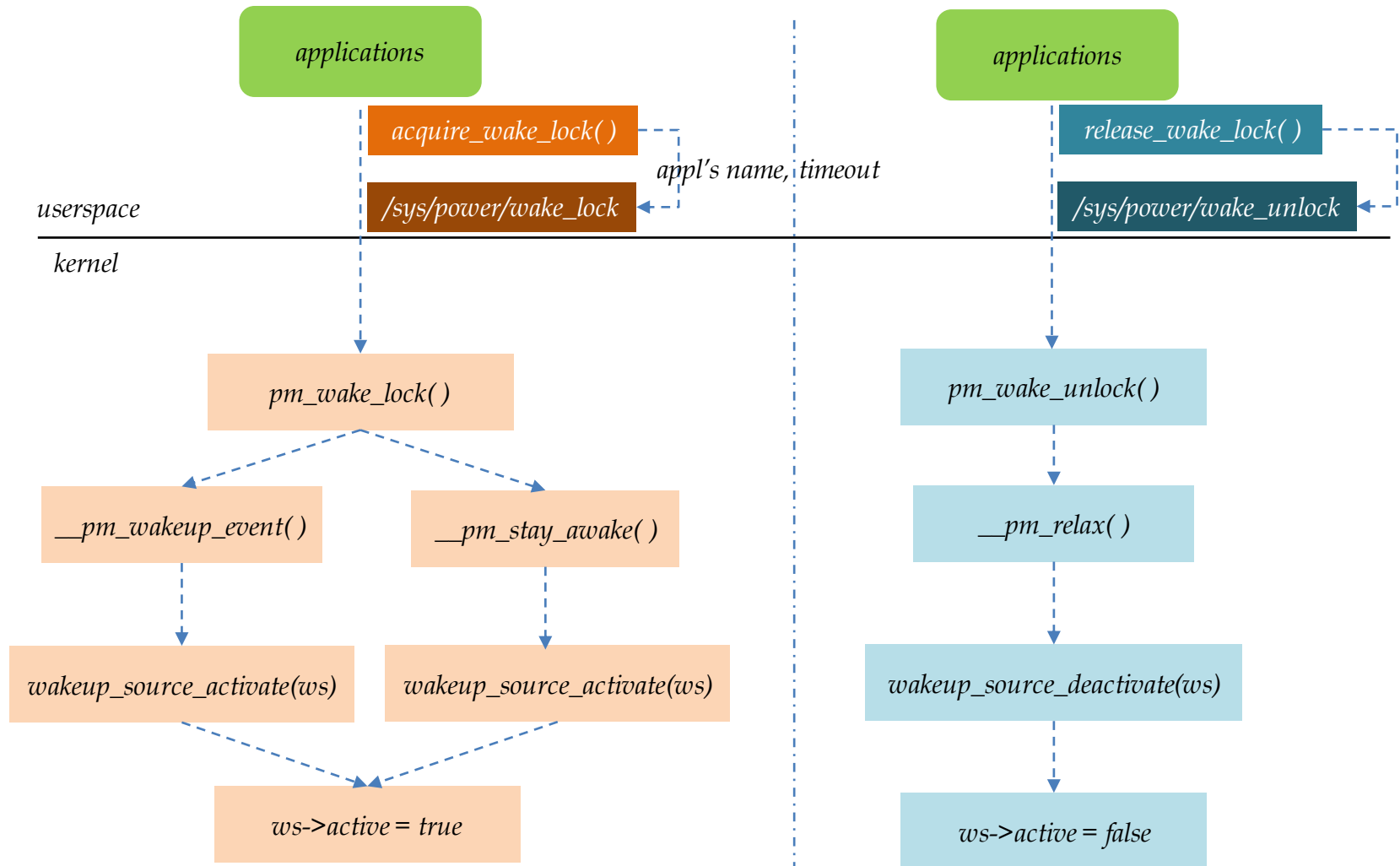
4. PM New Mechanism(2) – Wakeup Sources for Drivers



4. PM New Mechanism(3) – *Auto Sleep*



4. PM New Mechanism(4) - Wakelocks for Userspace



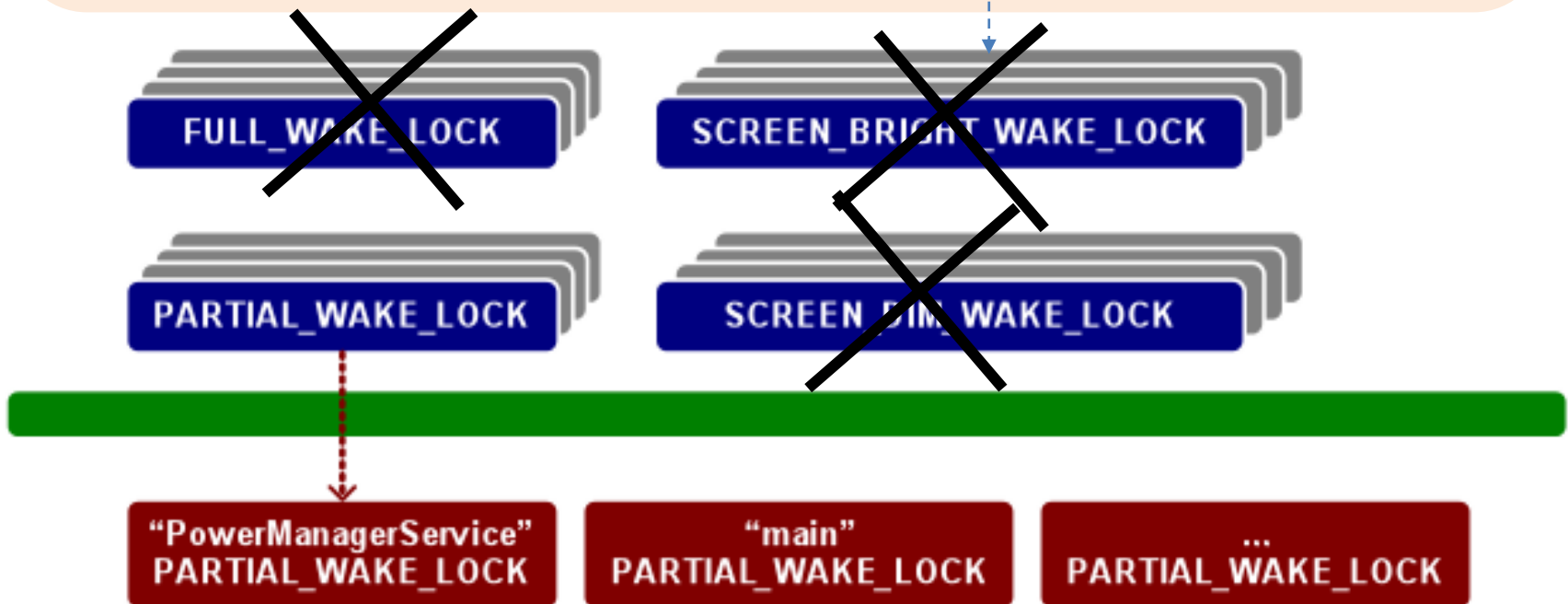
(*) 기존 android wakelock과의 호환을 위하여 user space용 wakelock API를 마련해 둬.
→ 내부 루틴은 wakeup source를 통하여 구현함.
→ Android area에는 partial wake_lock acquire & release만 존재하도록 변경됨.

5. Android Wakelock(1)

(*) 아래 그림은 최신 android의 그것과 내용이 다소 거리가 있을 수 있음.

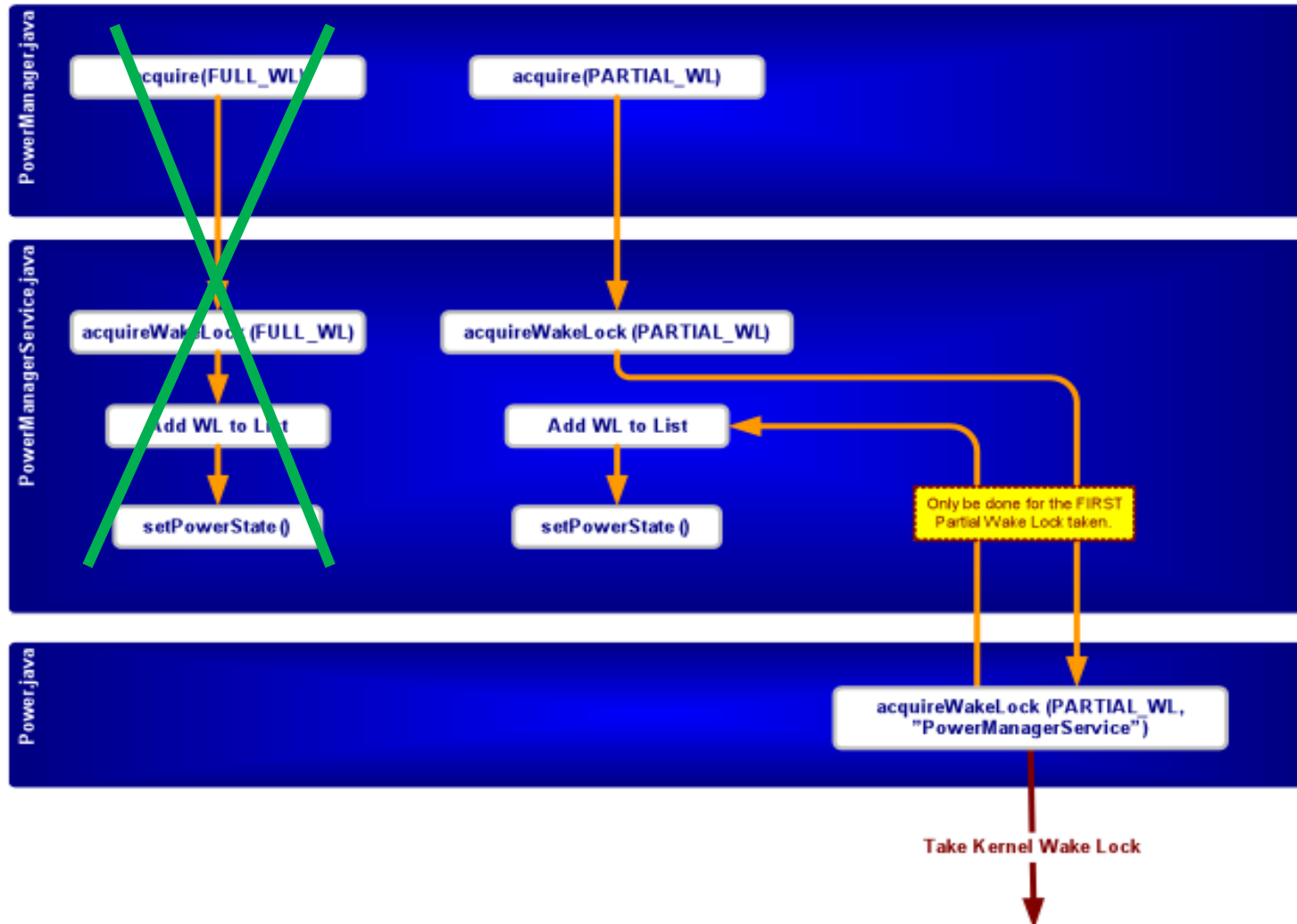
```
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);  
PowerManager.WakeLock wl =  
    pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "My  
Tag");
```

```
wl.acquire();  
/* screen will stay on during this section ... */  
wl.release();
```

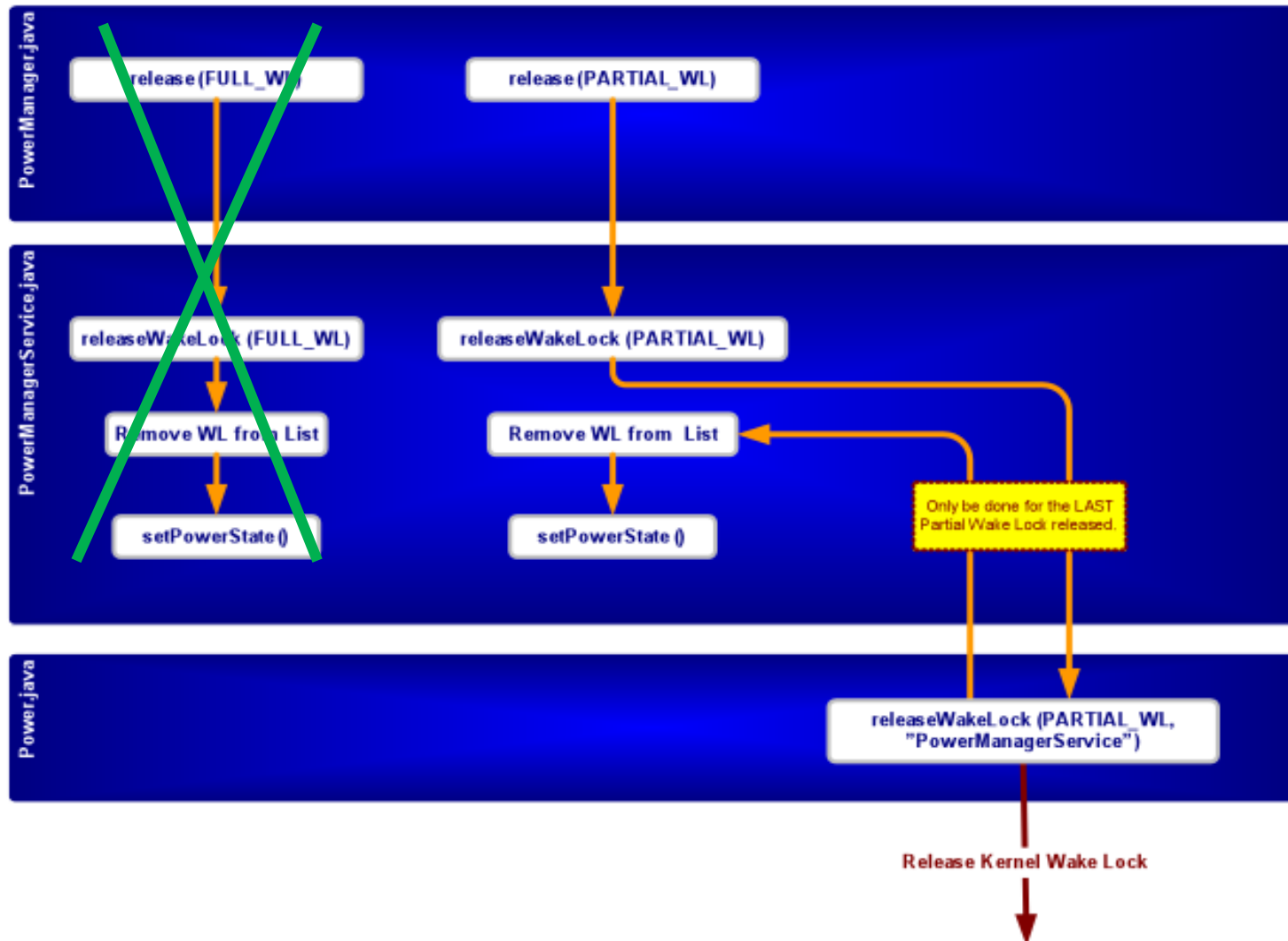


5. Android Wakelock(2) – *acquire wakelock*

(*) full wakelock은 기능에서 제거되었으며, partial wakelock만이 남아 있음.
→ 아래 그림은 최신 android의 그것과 내용이 다소 차이가 날 수 있음.

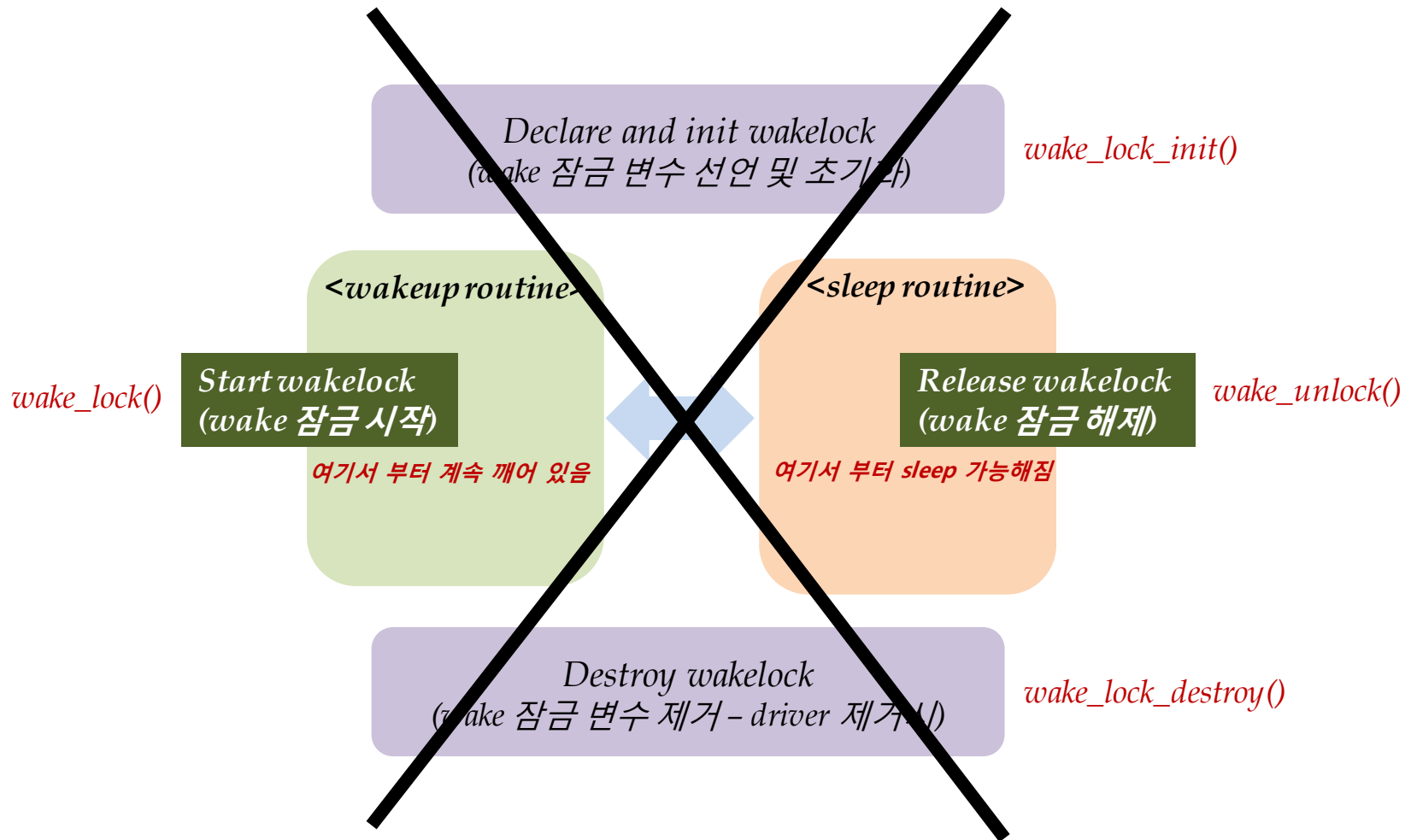


5. Android Wakelock(3) – *release wakelock*

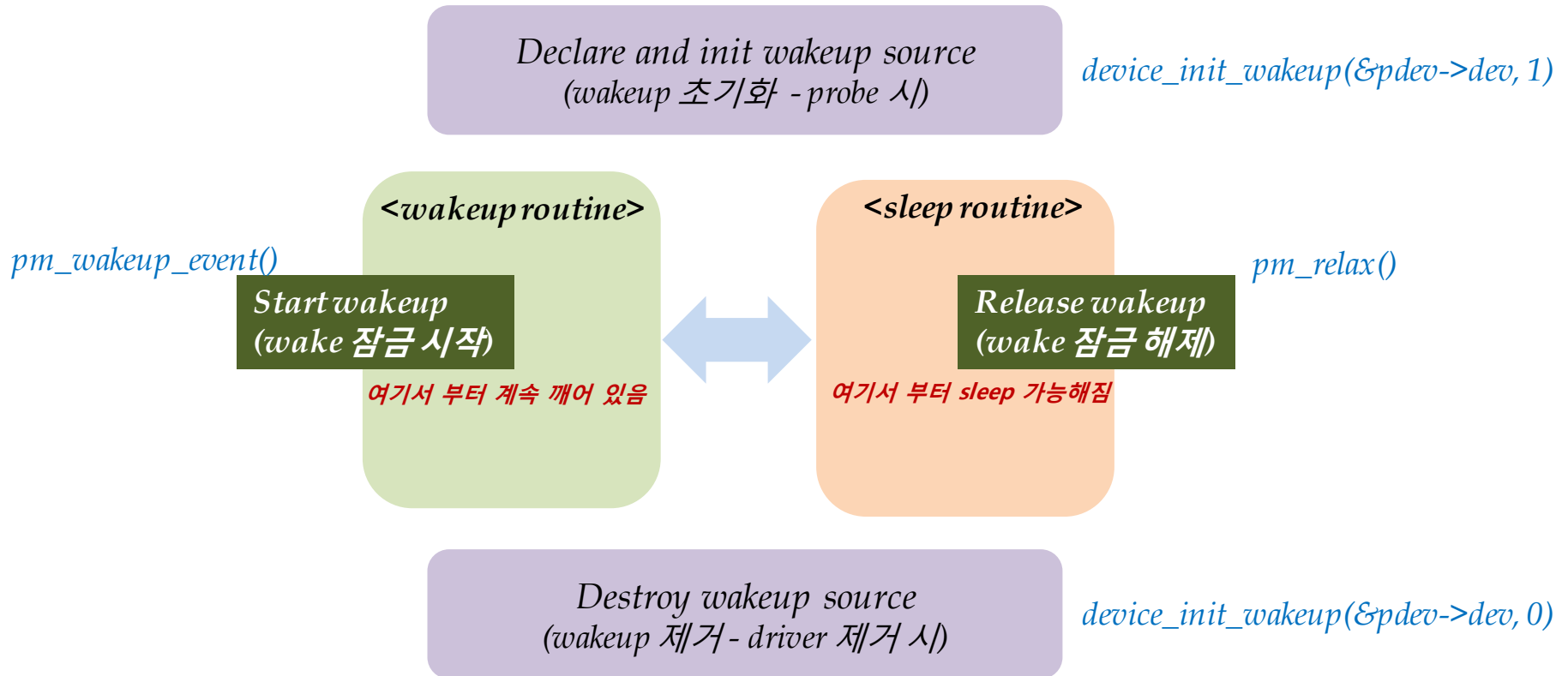


6. Wakeup Source vs Wakelock(1) – OLD kernel wakelock

(*) 기존 방식인 kernel wakelock은 다음 page의 wakeup source API로 교체되었음.



6. Wakeup Source vs Wakelock(2) – Wakeup Source



(*) `device_init_wakeup()` 호출 후, `sleep`(기존 `wake_unlock`) 혹은 `wakeup`(`wake_lock`) 진입 시점에 `pm_relax()`, `pm_wakeup_event()` 함수를 각각 호출해 주면 된다. 이는 기존에 `wakelock` 함수를 사용하던 위치와 동일함.

(*) `pm_stay_awake()` 함수는 wakeup 상태를 계속 유지하고자 할 경우에 사용함.

6. Wakeup Source vs Wakelock(3) – Wakeup Source API

```
extern void wakeup_source_prepare(struct wakeup_source *ws, const char *name);
extern struct wakeup_source *wakeup_source_create(const char *name);
extern void wakeup_source_drop(struct wakeup_source *ws);
extern void wakeup_source_destroy(struct wakeup_source *ws);
extern void wakeup_source_add(struct wakeup_source *ws);
extern void wakeup_source_remove(struct wakeup_source *ws);
extern struct wakeup_source *wakeup_source_register(const char *name);
extern void wakeup_source_unregister(struct wakeup_source *ws);
```

→ wakeup source internal 함수

```
extern int device_wakeup_enable(struct device *dev);
extern int device_wakeup_disable(struct device *dev);
extern void device_set_wakeup_capable(struct device *dev, bool capable);
extern int device_init_wakeup(struct device *dev, bool val);
extern int device_set_wakeup_enable(struct device *dev, bool enable);
```

→ 초기화 함수

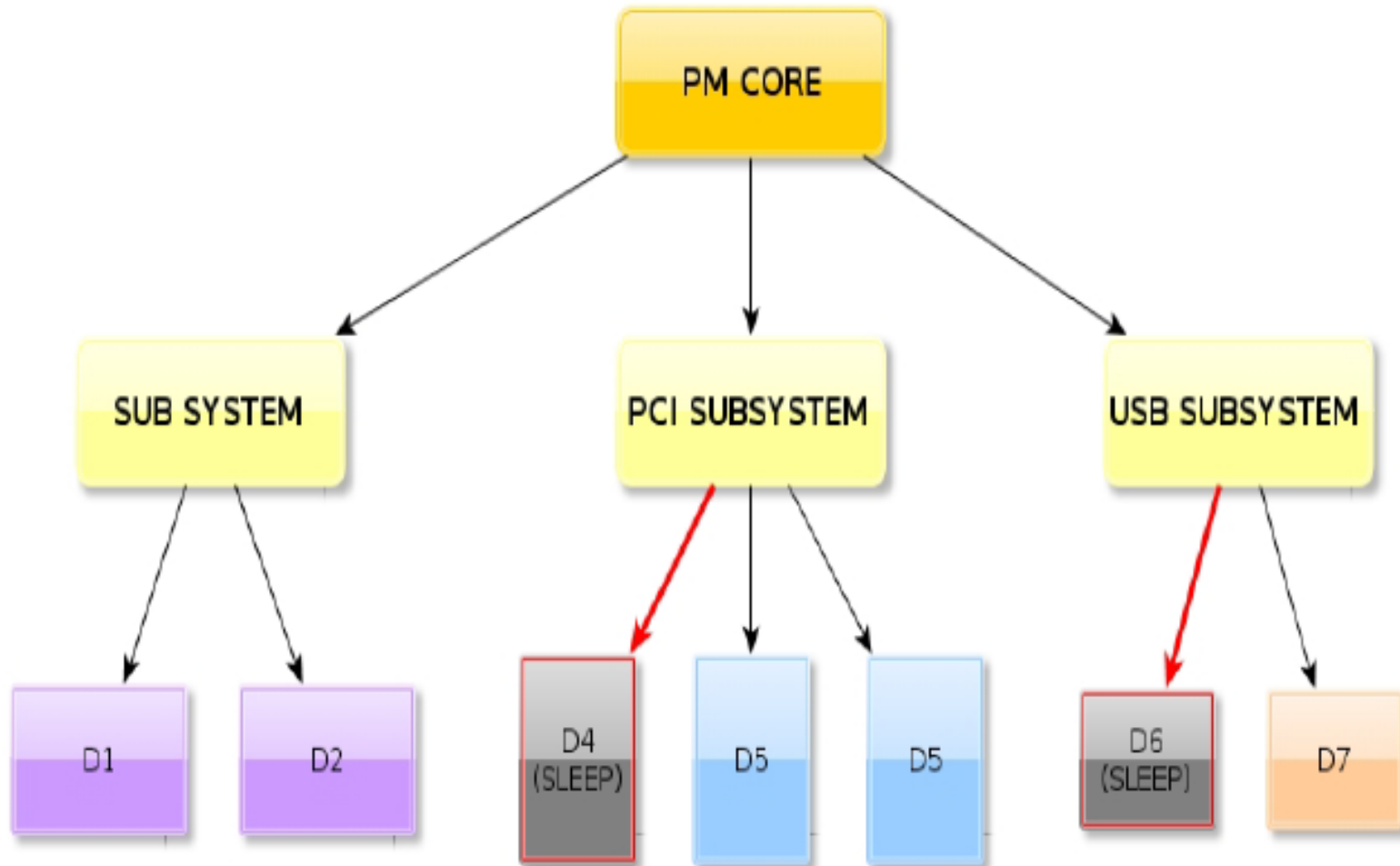
```
extern void __pm_stay_awake(struct wakeup_source *ws);
extern void pm_stay_awake(struct device *dev);
extern void __pm_relax(struct wakeup_source *ws);
extern void pm_relax(struct device *dev);
extern void __pm_wakeup_event(struct wakeup_source *ws, unsigned int msec);
extern void pm_wakeup_event(struct device *dev, unsigned int msec);
```

→ sleep/wakeup event 전달 함수

Runtime Power Management

7. Linux Runtime PM Overview(1)

(*) runtime PM은 개별 driver 마다의 sleep/wakeup과 관련이 있다.



7. Linux Runtime PM Overview(2) - 필요성

- System sleep is not enough to decrease runtime energy consumption
→ System sleep 만으로는 한계가 있음.
- Devices may depend on another device
→ Device 간에는 상호 의존 관계가 있음.
- Be helpful to figure out 'idle' condition
→ Idle 조건을 이해할 필요가 있음.
- Not doable to do I/O runtime PM in CPUIdle
→ CPU PM과 I/O device PM이 서로 다르게 운용될 수 있는 상황이 존재.
 - devices may be idle but cpu is not idle
 - task schedule may be caused during device suspend
 - some devices' suspend is moved from cpuidle platform driver to runtime PM (such as, uart suspend on omap3/4)
 - long latency is involved when returning from cpuidle

(*) runtime PM이 필요한 이유는 system suspend(정해진 순간에 일괄적으로 진행)에 한계 즉, System suspend 중에 특정 device가 suspend 조건에 부합하지 못하여, system이 suspend 상태로 진입하지 못하는 문제를 해결하기 위해 도입되었음.

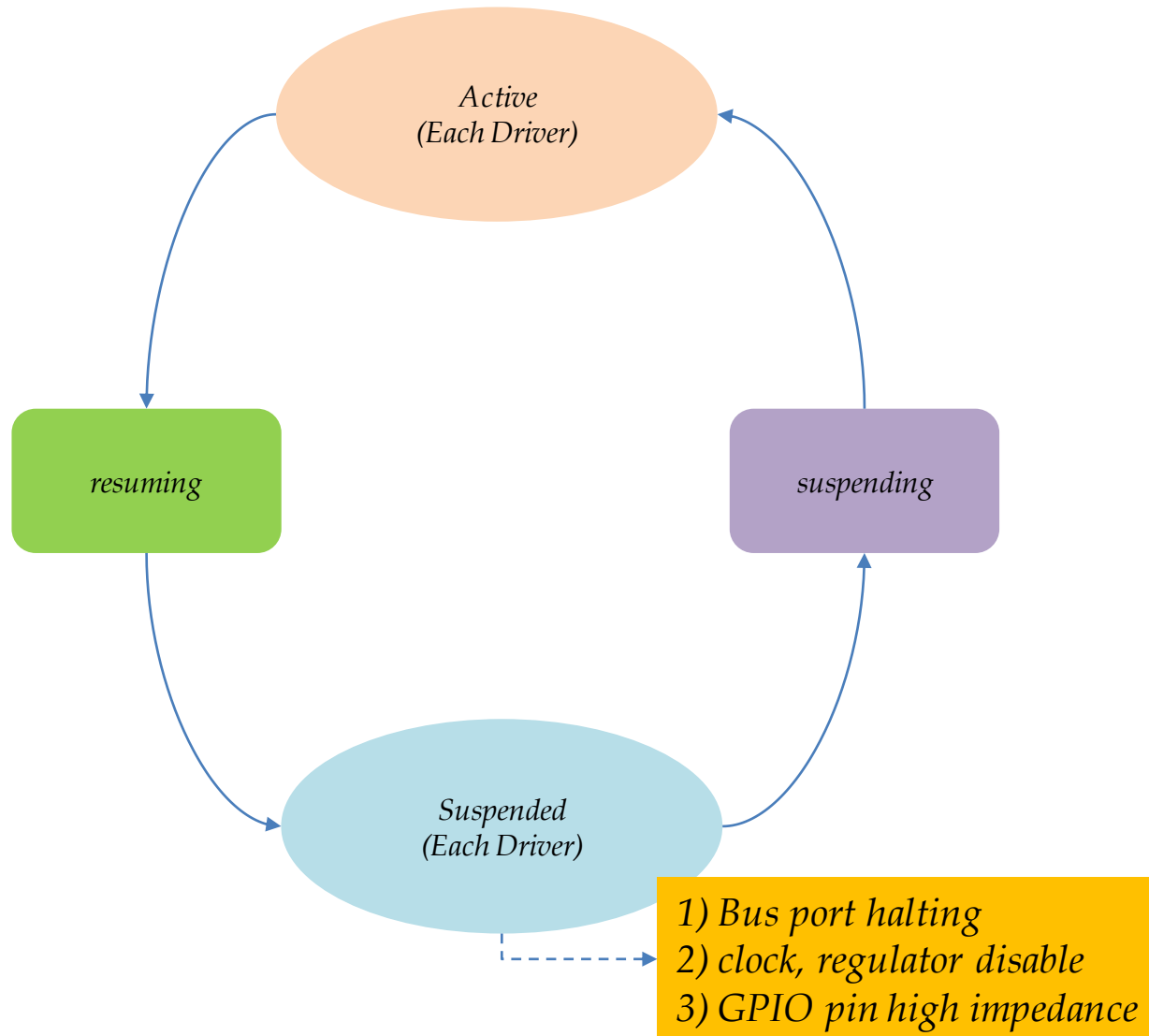
7. Linux Runtime PM Overview(3) - Runtime PM driver 사용 예

- *Usb subsystem HUB, usb mass storage, UVC, HID, CDC, serial, usb net, ...*
- *PCI subsystem e1000e, rtl8169, ehci-pci, uhci-pci, ...*
- *SCSI subsystem sd*
- *I2C subsystem*
- *MMC subsystem*
- *Serial devices*
- *Misc device(gpio, key,.....)*
- *ARCHs(RPM via dev_pm_domain)*

() Runtime PM은 CPU를 위한 것이 아니라, I/O device를 위해 마련된 기법이다.*

8. Linux Runtime PM State Diagram

(*) runtime PM은 개별 driver의 suspend/resume과 연관이 있음.



9. Linux Runtime PM Callbacks

(*) 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 작업을 수행한다.

(*) `include/linux/pm.h` 파일 참조

```
struct dev_pm_ops {  
    int (*suspend)(struct device *dev);  
    int (*resume)(struct device *dev);  
    ...  
    int (*runtime_suspend)(struct device *dev);  
    int (*runtime_resume)(struct device *dev);  
    int (*runtime_idle)(struct device *dev);  
    ...  
};
```

- `->runtime_suspend()`
 - Save context
 - Power down HW
- `->runtime_resume()`
 - Power up HW
 - Restore context
- `->runtime_idle()`

(*) Runtime PM을 사용하기 위해서는 `CONFIG_PM_RUNTIME=y` 이 켜져 있어야 함.

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

10. Linux Runtime PM API 사용법(1)

- Probe
 - `pm_runtime_enable()`
 - probe/configure hardware
 - `pm_runtime_suspend()`
- Activity
 - `pm_runtime_get()`
 - Do work
 - `pm_runtime_put()`
- Done
 - `pm_runtime_suspend()`

10. Linux Runtime PM API 사용법(2)

- I need the device

- `pm_runtime_get(), _sync(), _noresume()`
- Increment use count, `pm_runtime_resume()`

- I'm done

- `pm_runtime_put(), _sync, _noidle()`
- Decrement use count, `pm_runtime_idle()`

10. Linux Runtime PM API 사용법(3)

1) Probe 단계

```
pm_runtime_enable()  
/* probe/configure hardware .. */  
pm_runtime_suspend()
```

(*) remove() 함수에서는 pm_runtime_disable()
함수를 호출함.

2) Activity 단계(실제 코드 내에서)

```
pm_runtime_get()  
/* do work */  
pm_runtime_put()
```

3) Suspend에 진입하고자 할 때

```
pm_runtime_suspend()
```

4) Runtime callback 호출

a) pm_runtime_suspend(dev),
pm_schedule_suspend(dev, delay)
device can suspend subsys: ->runtime_suspend()
driver: -->runtime_suspend()

b) pm_runtime_resume(dev), pm_request_resume(dev)
subsys: -->runtime_resume()
subsys: -->runtime_idle()
driver: -->runtime_resume()

c) pm_runtime_idle(dev), pm_request_idle(dev)
driver: -->runtime_idle()

5) Device를 사용할 경우 및 사용을 끝낸 경우 사용할 함수

a) 사용하고자 할 경우 호출하는 함수
pm_runtime_get(), _sync(), _noresume() Increment
use count, pm_runtime_resume()
⇒ device를 resume해 줌.

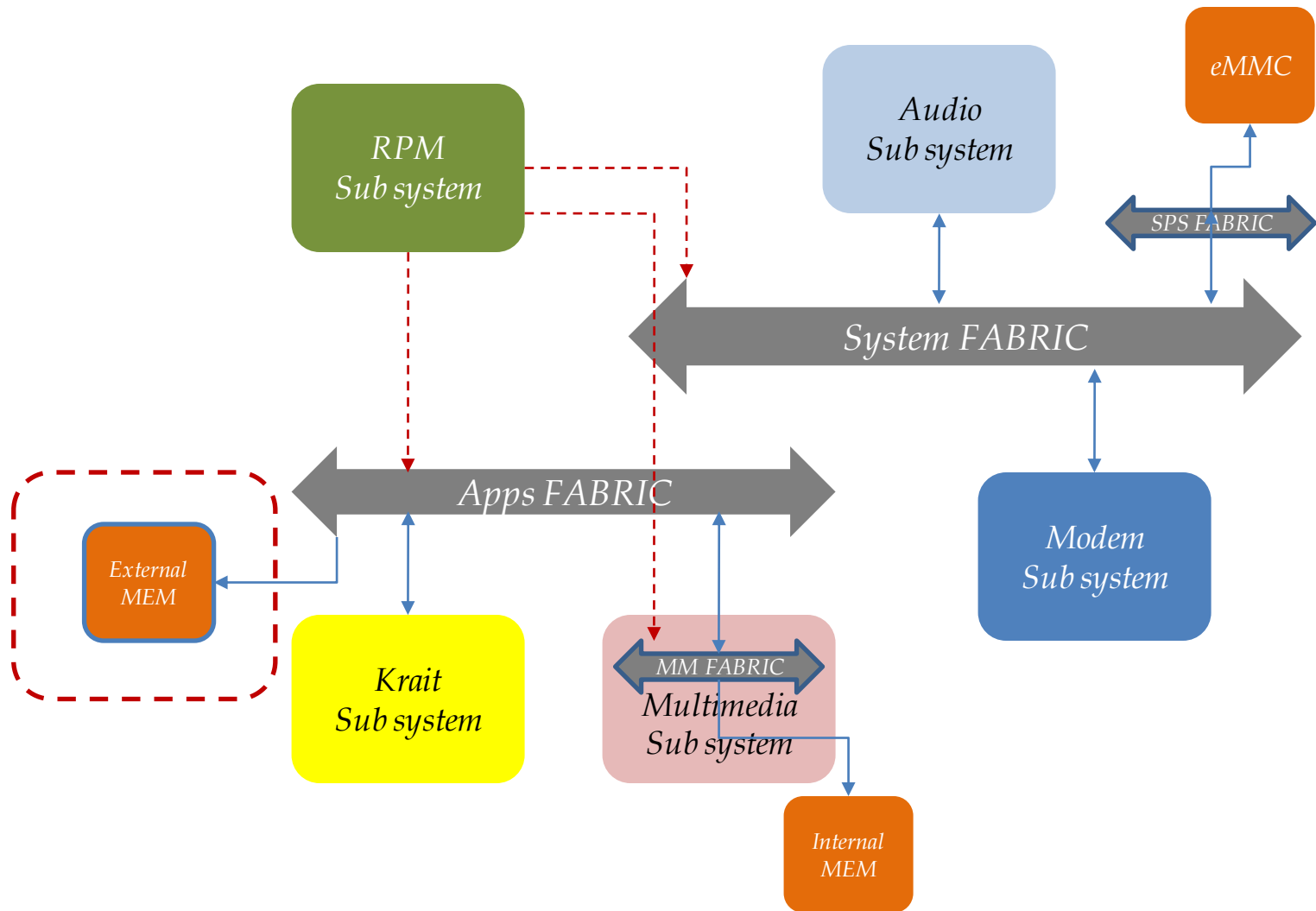
b) 사용을 다한 경우 호출하는 함수
pm_runtime_put(), _sync, _noidle() Decrement use
count, pm_runtime_idle()
⇒ device를 idle 상태로 만듦.

10. Linux Runtime PM API 사용법(4)

```
static inline int pm_runtime_idle(struct device *dev)
static inline int pm_runtime_suspend(struct device *dev)
static inline int pm_runtime_autosuspend(struct device *dev)
static inline int pm_runtime_resume(struct device *dev)
static inline int pm_request_idle(struct device *dev)
static inline int pm_request_resume(struct device *dev)
static inline int pm_request_autosuspend(struct device *dev)
static inline int pm_runtime_get(struct device *dev)
static inline int pm_runtime_get_sync(struct device *dev)
static inline int pm_runtime_put(struct device *dev)
static inline int pm_runtime_put_autosuspend(struct device *dev)
static inline int pm_runtime_put_sync(struct device *dev)
static inline int pm_runtime_put_sync_suspend(struct device *dev)
static inline int pm_runtime_put_sync_autosuspend(struct device *dev)
static inline int pm_runtime_set_active(struct device *dev)
static inline void pm_runtime_set_suspended(struct device *dev)
static inline void pm_runtime_disable(struct device *dev)
static inline void pm_runtime_use_autosuspend(struct device *dev)
static inline void pm_runtime_dont_use_autosuspend(struct device *dev)
```

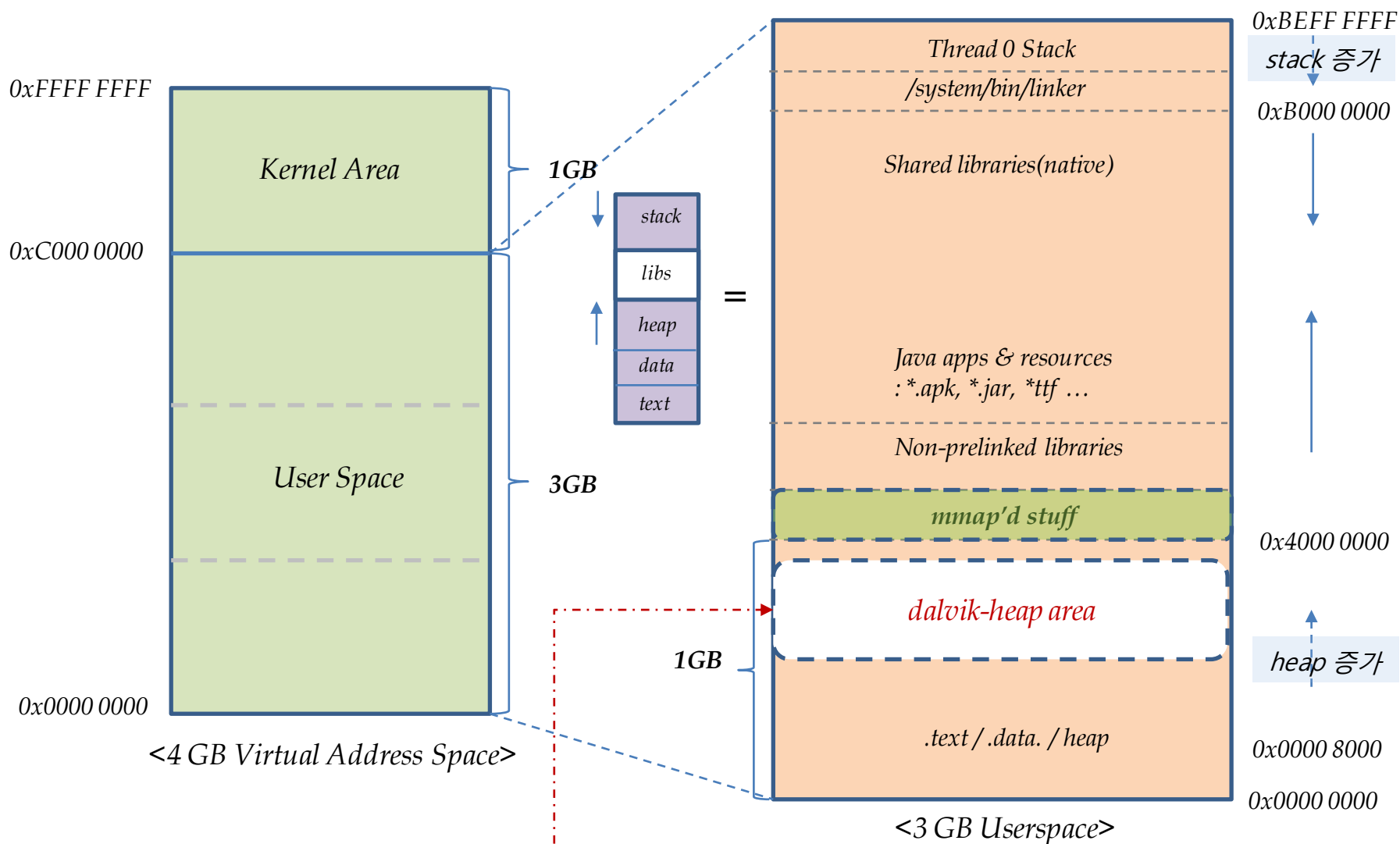
Memory Management

1. Memory Devices



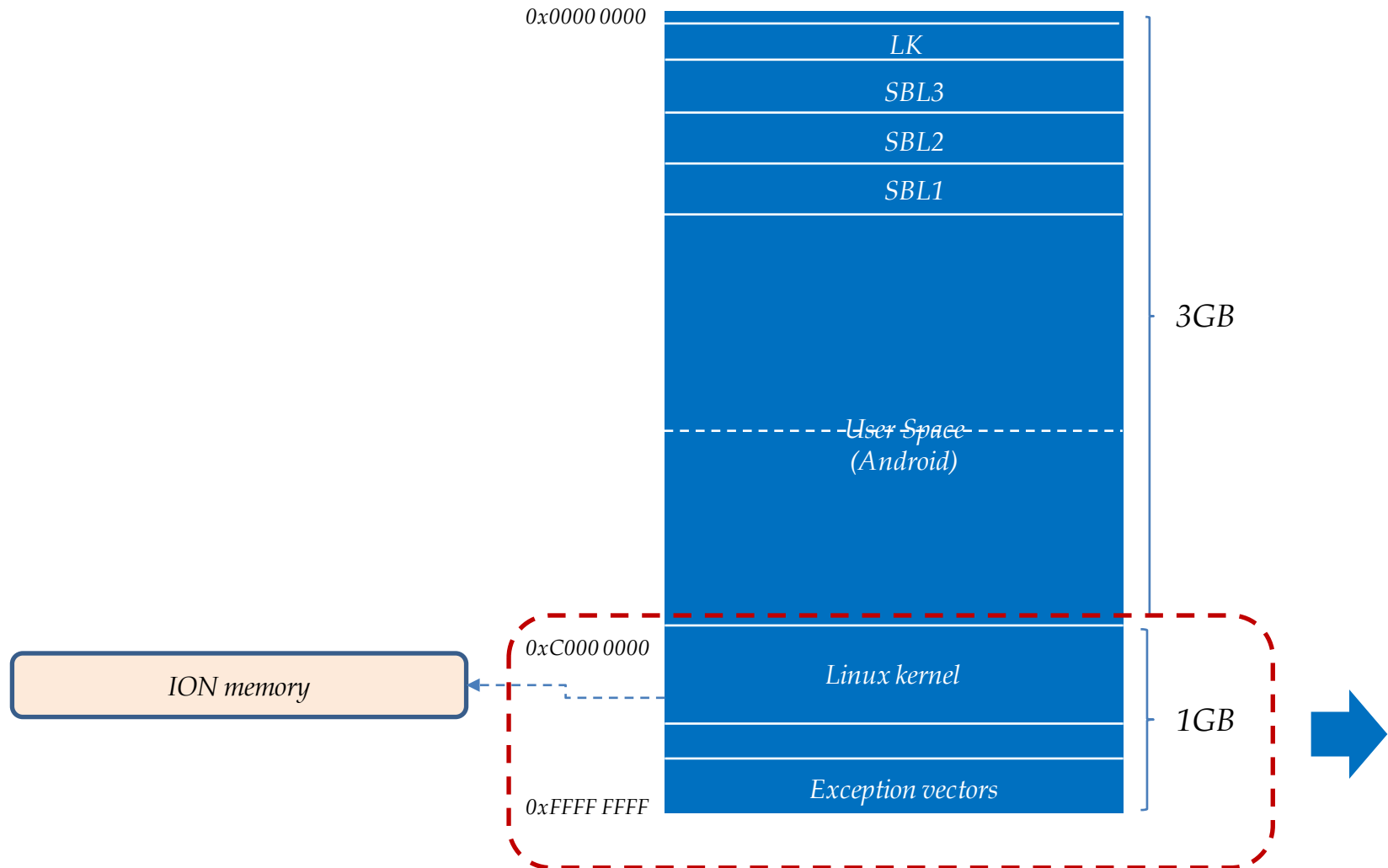
2. Virtual Memory Map(1) - *android(userspace) memory map*

(*) 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)

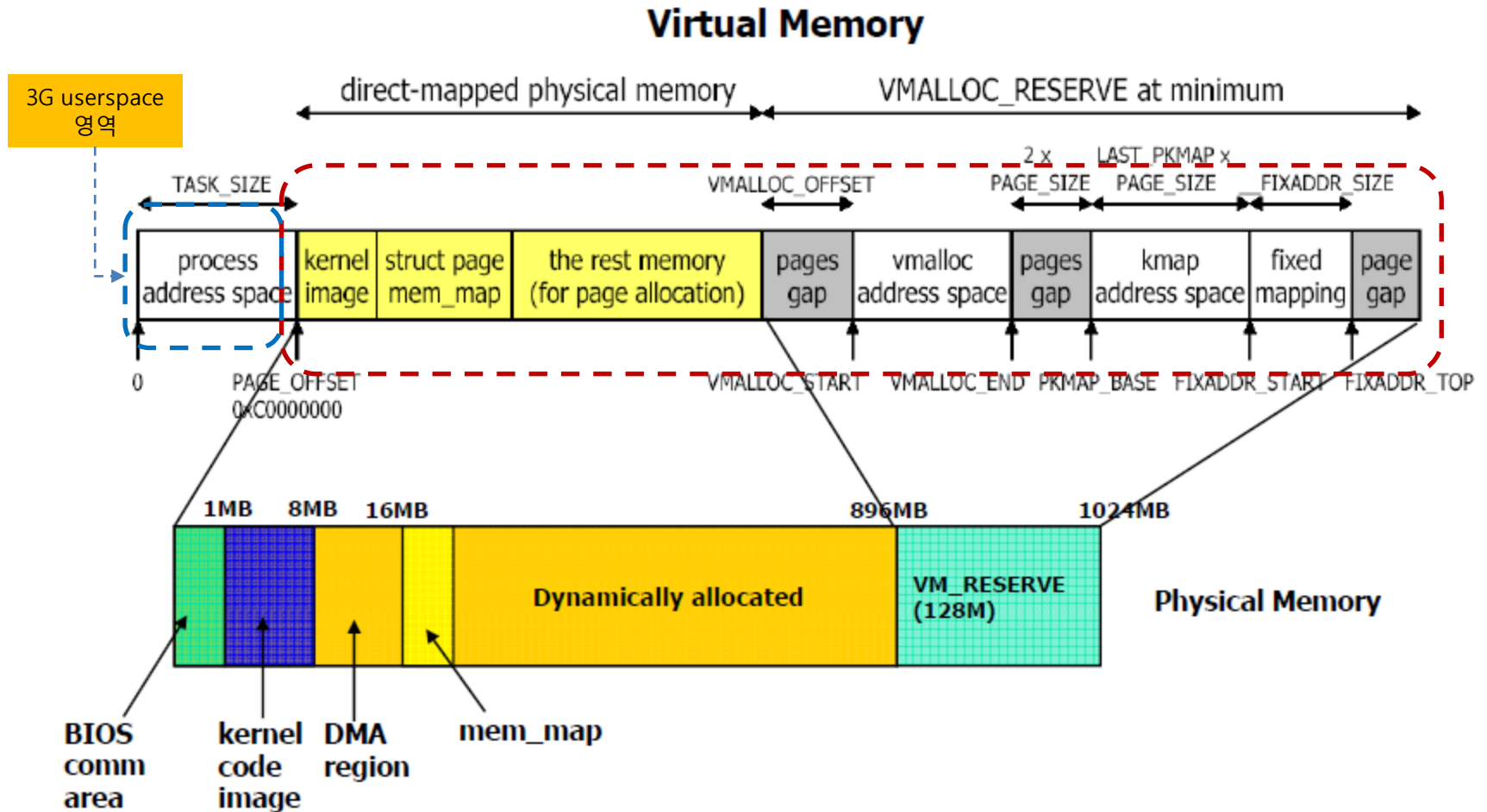


Dalvik VM에서 할당하는 heap의 위치는 추측일 뿐임^^

2. Virtual Memory Map(2) – *kernel map/1*



2. Virtual Memory Map(3) – *kernel map/2*



3. Slub Allocator - *SLUB Debug*

- **1) config 설정**

- **CONFIG_SLUB_DEBUG=y**

- *General setup ->*

- *[*] Enable SLUB debugging support*

- **CONFIG_SLUB_DEBUG_ON=y**

- *Kernel hacking ->*

- *[*] SLUB debugging on by default*

←이걸 켜셔야 실제로 memory 문제를 잡을 수 있음.

- **2) SLUB_DEBUG_ON 관련 부연 설명**

- CONFIG_SLUB_DEBUG_ON feature를 켜시면, 아래 내용이 enable되게 되며, 성능 저하(?)가 있을 수 있음.

- 또한, 아래 feature 중, Red Zone(Z)을 켜시면 부팅이 안될 수도 있겠음...

- =====

- kernel/mm/slub.c

- **#define DEBUG_DEFAULT_FLAGS (SLAB_DEBUG_FREE | SLAB_RED_ZONE | W**

- **SLAB_POISON | SLAB_STORE_USER)**

- =====

- **3) SLUB_DEBUG_ON 과 관련된 flag 설명**

- F Sanity checks on (enables SLAB_DEBUG_FREE. Sorry

- → 주로 slub memory free 시 오류(가령, double free)를 감지하고, 수정

- Z Red zoning

- → 할당 영역 앞/뒤에 red zone 추가하여 넘치는 문제 감지 및 수정

- P Poisoning (object and padding)

- → 초기화 안된 상태에서 할당된 영역을 사용하는 것을 방지하기 위해 poison(a5a5a5a5) 추가

- =====

- [38](#) F Sanity checks on (enables SLAB_DEBUG_FREE. Sorry

- [39](#) SLAB legacy issues)

- [40](#) Z Red zoning

- [41](#) P Poisoning (object and padding)

- [42](#) U User tracking (free and alloc)

- [43](#) T Trace (please only use on single slabs)

- [44](#) A Toggle failslab filter mark for the cache

- [45](#) O Switch debugging off for caches that would have

- [46](#) caused higher minimum slab orders

- [47](#) - Switch all debugging off (useful if the kernel is

- [48](#) configured with CONFIG_SLUB_DEBUG_ON)

- =====

- Slub debug 관련 보다 자세한 사항은 kernel/Documentation/vm/slub.txt 파일 참조 !

4. Android Driver - ION memory(1)

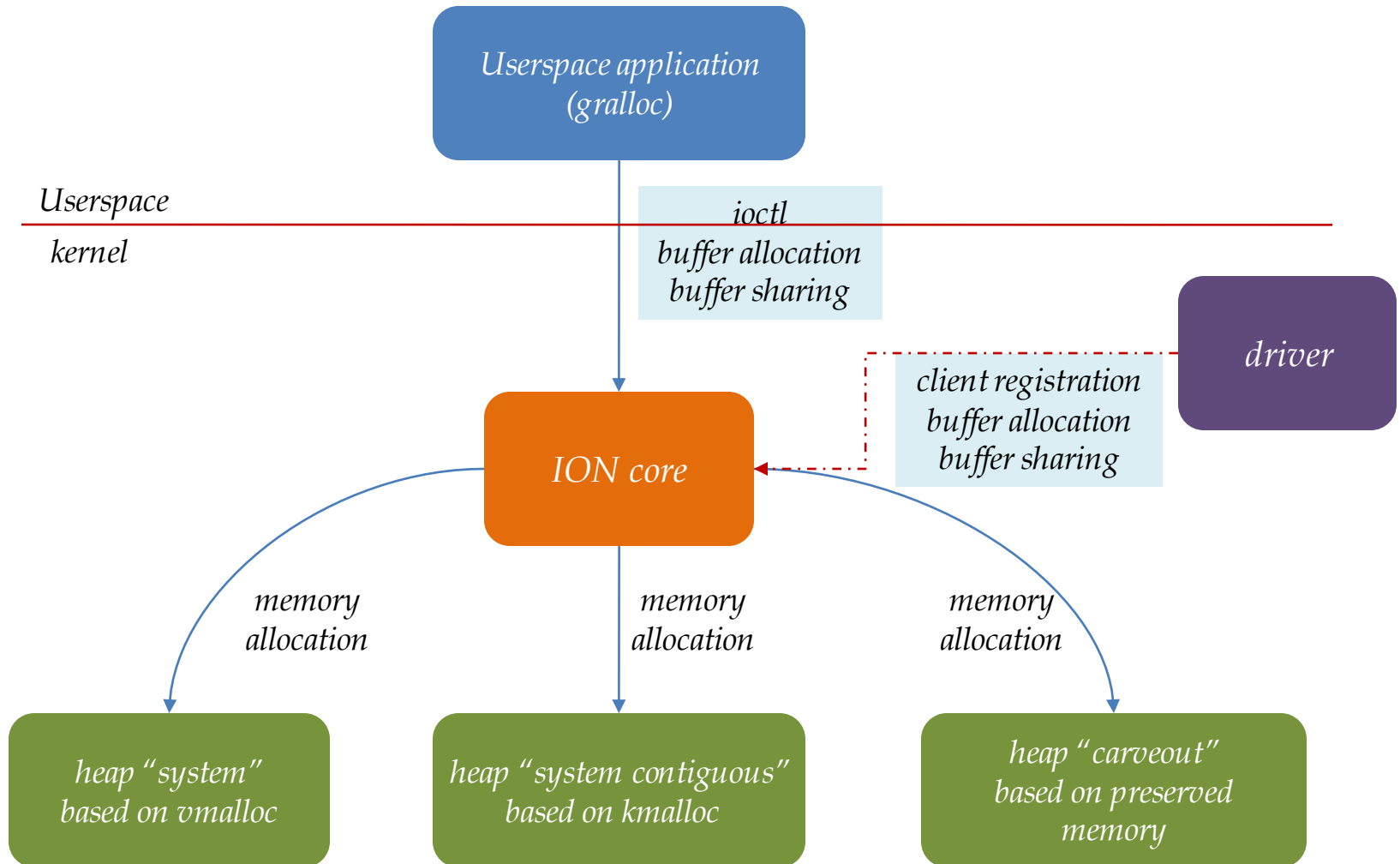
<ION memory 개념 소개/1>

- 1) ICS version 부터 Google이 만듦.
- 2) large contiguous memory 를 유지/관리하기 위해 만듦.
→ Physical contiguous memory 뿐 아니라, virtual contiguous memory도 지원함.
- 3) Process 간의 memory 공유를 위해 binder 등과도 밀접히 연관되어 있음.
- 4) 기존에 있던 유사한 개념
 - NVIDIA Tegra - NVMAP
 - TI OMAP - CMEM
 - Qualcomm MSM - PMEM→ ION으로 옮겨 가고 있음

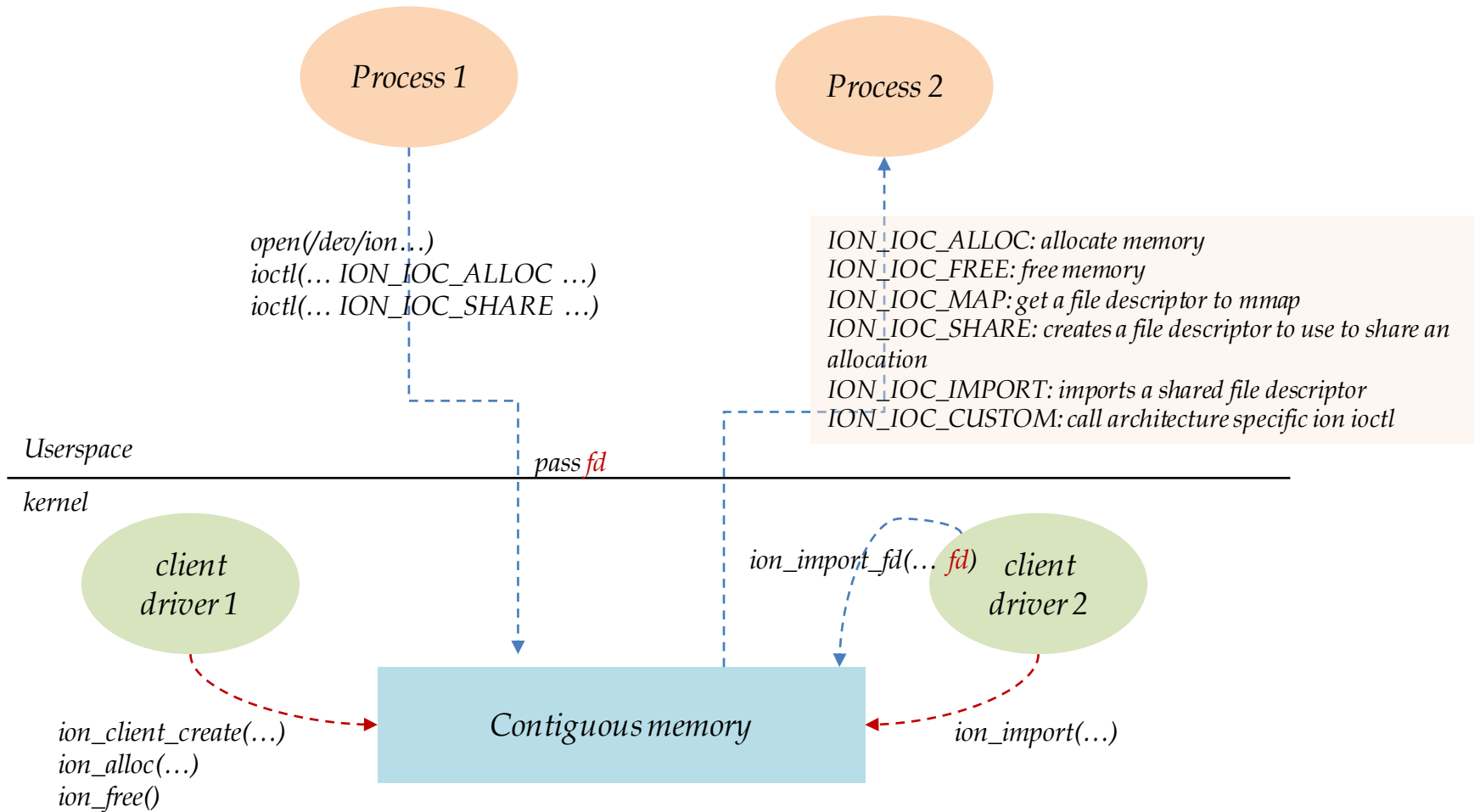
<ION memory 개념 소개/2>

- 1) ION은 하나 이상의 memory pool을 관리함
- 2) fragmentation 문제를 처리하기 위해 고안되었으며, booting시에 할당됨.
- 3) GPU, display controller, camera 등이 주요 대상(special memory가 필요한 애들 !)
- 4) 관련 코드
 - `include/linux/ion.h`
 - `drivers/gpu/ion/*`

4. Android Driver - *ION* memory(2)



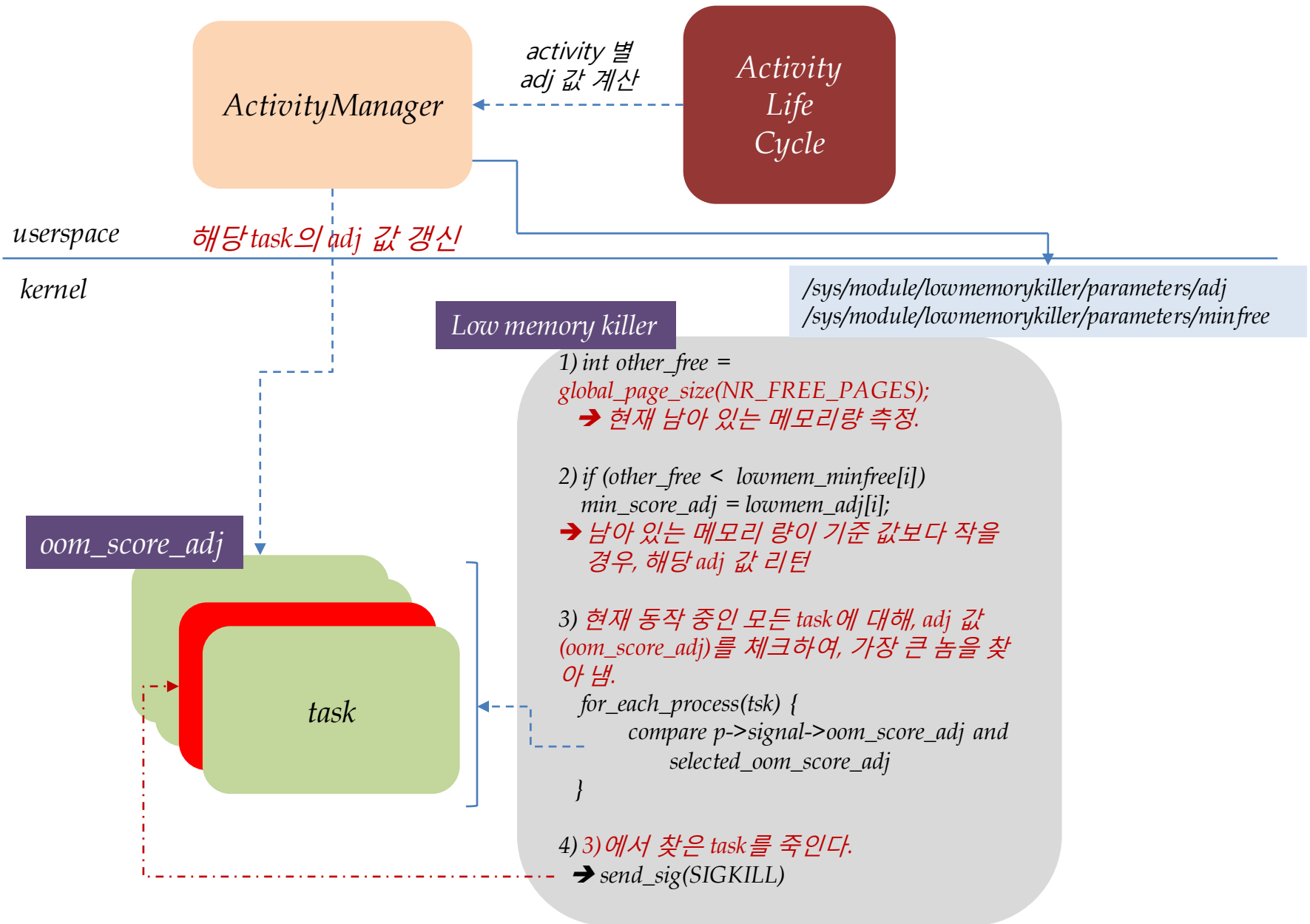
4. Android Driver - ION memory(3)



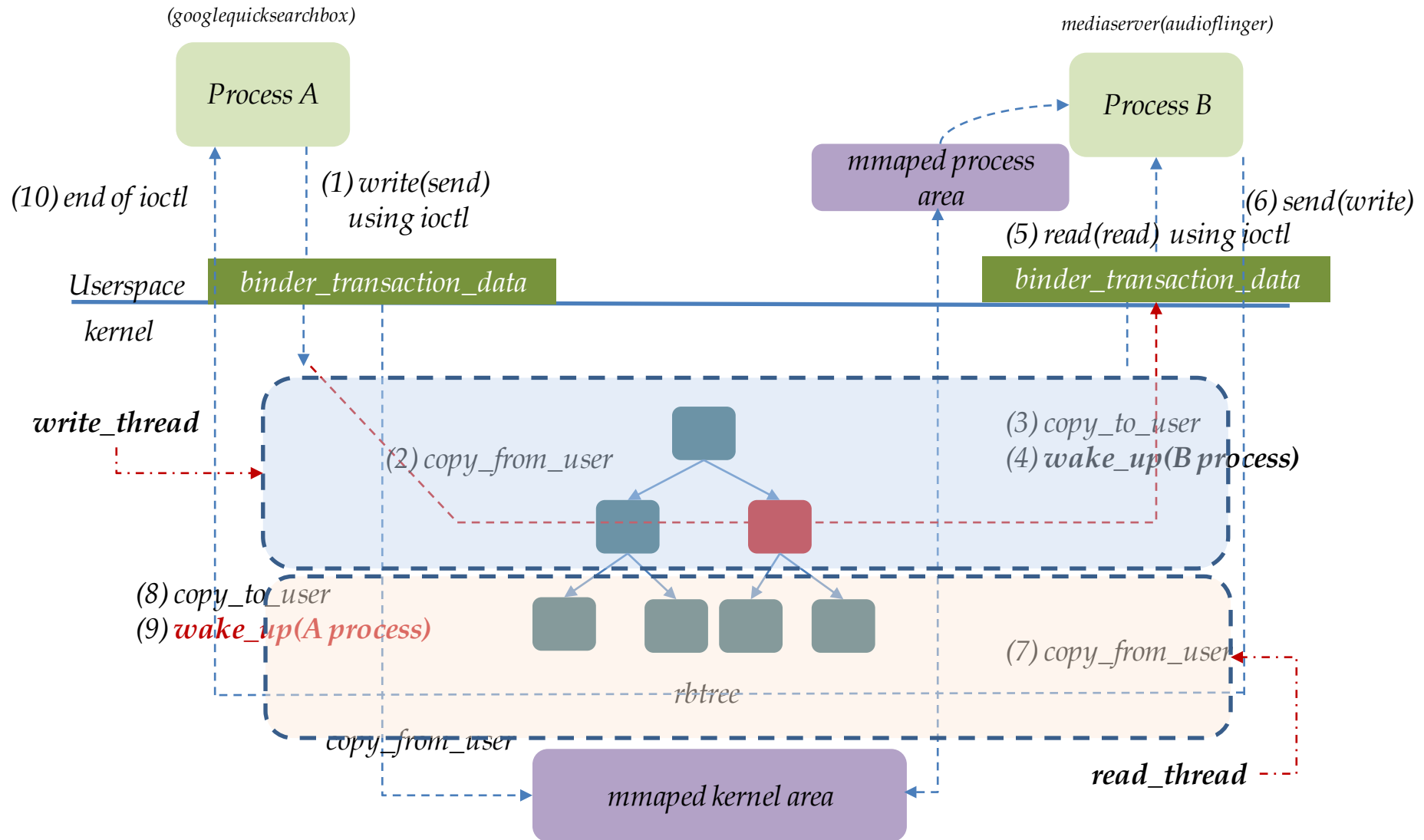
`ion_phys`: returns the physical address and len of a handle
`ion_map_kernel`: create mapping for the given handle
`ion_unmap_kernel`: destroy a kernel mapping for a handle
`ion_map_dma`: create a dma mapping for a given handle, return an sglist
`ion_unmap_dma`: destroy a dma mapping for a handle

`ion_share`: given a handle, obtain a buffer to pass to other clients
`ion_import`: given an buffer in another client, import it
`ion_import_fd`: given an fd obtained via `ION_IOC_SHARE` ioctl, import it

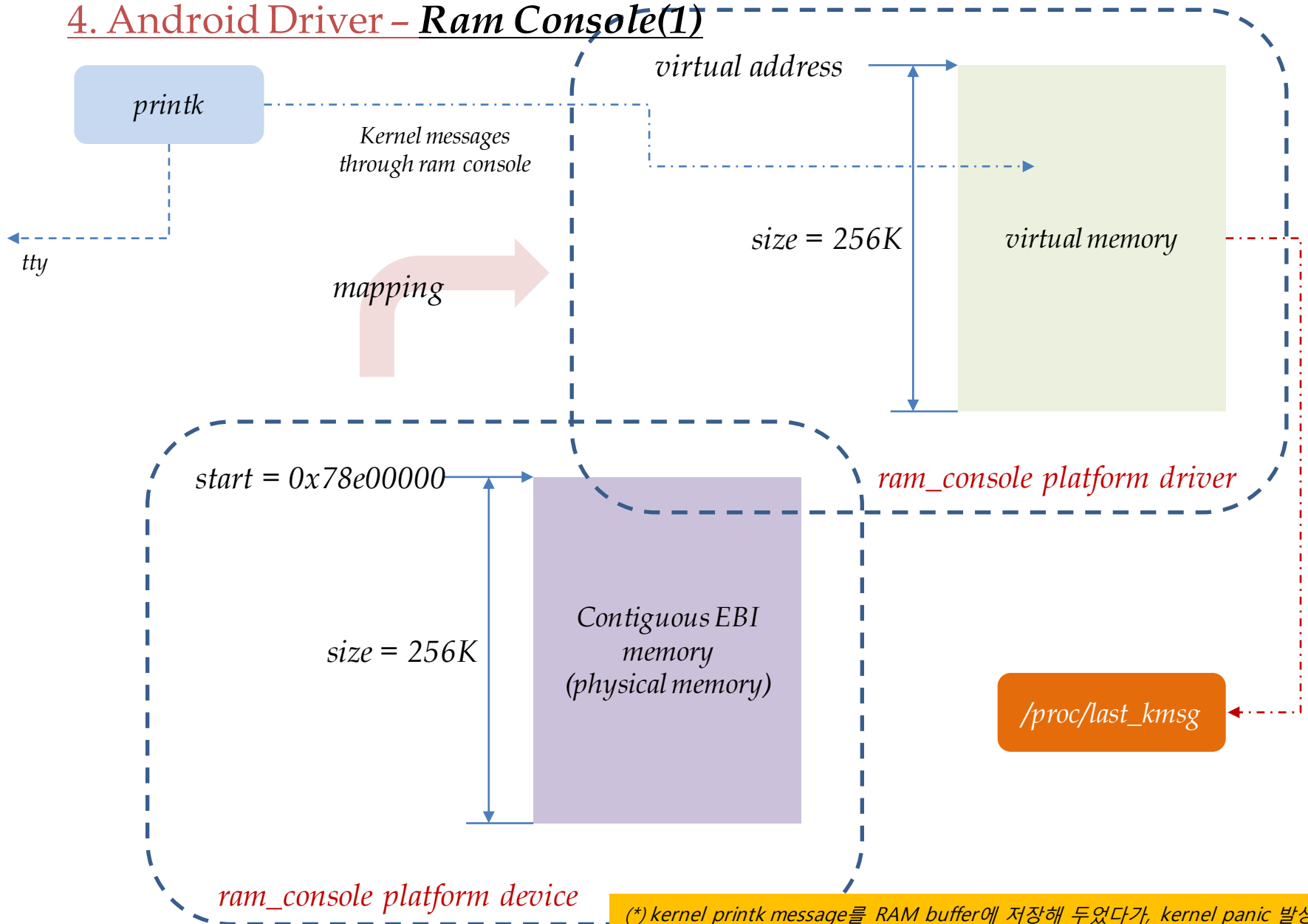
4. Android Driver: Lowmemorykiller



4. Android Driver – *binder & Ashmem*



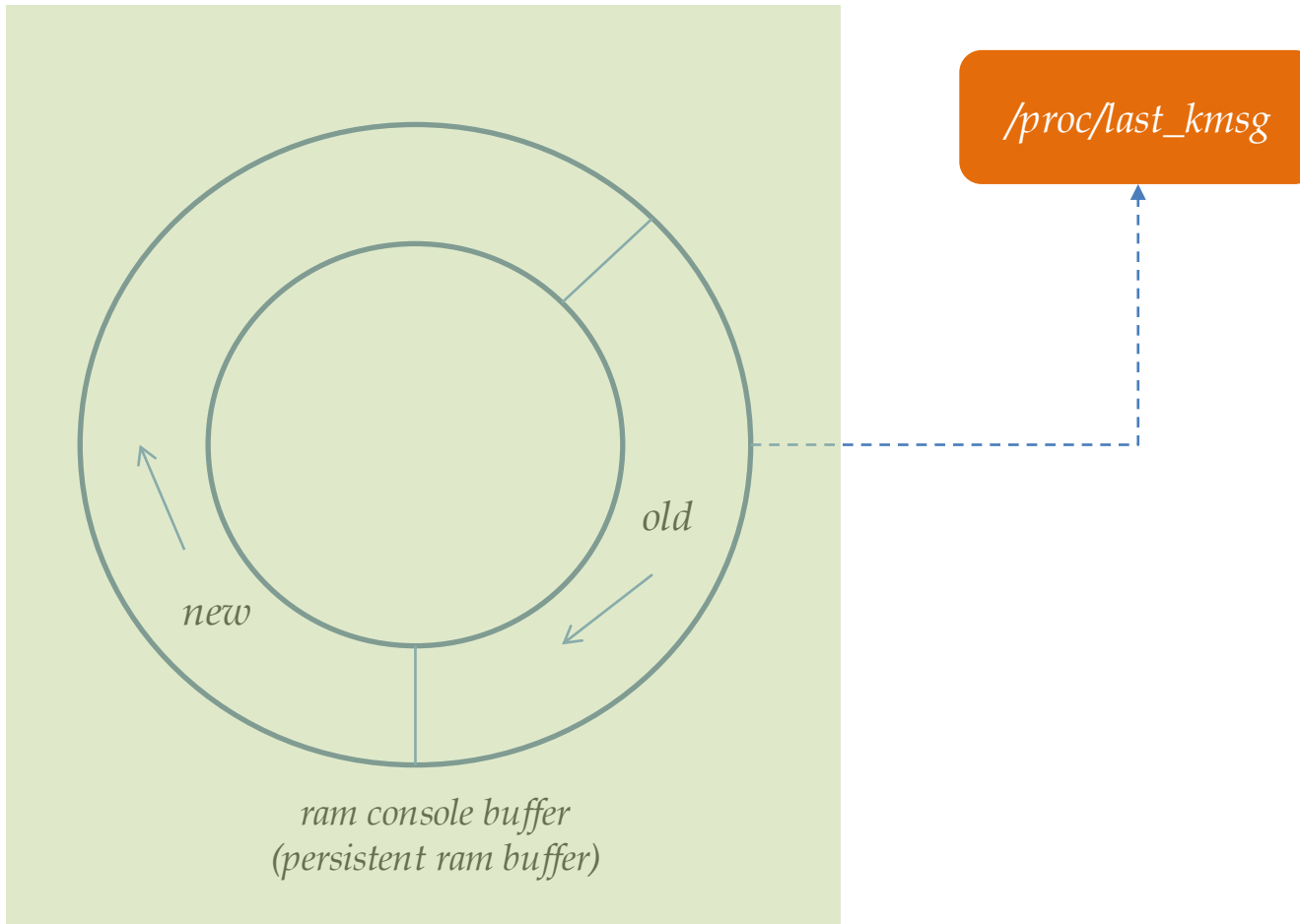
4. Android Driver – Ram Console(1)



(*) kernel `printk` message를 RAM buffer에 저장해 두었다가, kernel panic 발생 시, (다음 번 booting 시점에서) `/proc/last_kmsg`를 통해 이전 상태를 확인할 수 있도록 해주는 기능 → ram console !!!

4. Android Driver – *Ram Console(2)*

(*) # `cat /proc/last_kmsg` 하면, 이전 부팅 상태의 *kernel message*가 출력될 것이다^^.
→ 이전에 *kernel panic*이 발생했다면 ... 그 이유를 알 수 있음^^



Notifier

1. Notifier Concept(1)

(*) *notifier*는 서로 다른 곳에 위치한 *kernel code* 간에 *event*를 전송하기 위한 *mechanism*으로 *callback* 함수 개념으로 생각하면 이해가 쉽다^^.

(*) 즉, *kernel routine A*에서는 호출되기를 원하는 *callback* 함수를 기술 및 등록하고, *event* 발생 시점을 아는 *kernel routine B*에서 해당 함수를 호출해 주는 것으로 설명할 수 있겠음 !

<kernel routine A>

```
{
  notifier callback 함수 정의
  → struct notifier_block
  notifier callback 함수 등록
  → XXX_notifier_chain_register(&chain_name, nb)
  ...
}

my_callback_func()
{
}
}
```

<event 발생 시점 예 - bluetooth>

- 1) HCI_DEV_UP(link up 시)
- 2) HCI_DEV_DOWN(link down 시)
- 3) HCI_DEV_REG
- 4) HCI_DEV_UNREG
- 5) HCI_DEV_WRITE(패킷 전송 시)

<kernel routine B>

```
{
  notifier chain register

  ...
  notifier chain unregister
}

some_func()
{
  call notifier_callback func
  → XXX_notifier_call_chain(...)
  → 특정 event 발생 시점에서 호출
}
```

1. Notifier Concept(2) – *panic notifier list example*

2) notifier chain 등록

```
XXX_probe()
{
    ...
    atomic_notifier_chain_register(
        &panic_notifier_list, &test_notifier)
    ...
}

test_func()
{
    ...
}

...
```

실제 원하는 event 발생 시, notifier_call 호출 !

XXX platform driver

1) notifier block 정의

```
struct notifier_block test_notifier = {
    .notifier_call = test_func,
};
```

3) notifier chain 호출

```
panic()
{
    ...
    atomic_notifier_call_chain(
        &panic_notifier_list, ...)
    ...
}
```

1. Notifier Concept(3) - *notifier chain*의 종류

- 1) Atomic notifier chains
- 2) Blocking notifier chains
- 3) Raw notifier chains
- 4) SRCU(Sleepable Read-Copy Update) notifier chains

2. Notifier Data Structure

(*) 자세한 사항은 `include/linux/notifier.h` 파일 참조 !

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};

extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
    struct notifier_block *nb);
extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
    struct notifier_block *nb);
struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block *head;
};
extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
    struct notifier_block *nb);
extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
    struct notifier_block *nb);

struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block *head;
};
extern int blocking_notifier_chain_cond_register(
    struct blocking_notifier_head *nh,
    struct notifier_block *nb);
extern int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,
    struct notifier_block *nb);
extern int blocking_notifier_chain_unregister(struct blocking_notifier_head *nh,
    struct notifier_block *nb);
extern int raw_notifier_chain_unregister(struct raw_notifier_head *nh,
    struct notifier_block *nb);
extern int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,
    struct notifier_block *nb);

extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
    unsigned long val, void *v);
extern int __atomic_notifier_call_chain(struct atomic_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
    unsigned long val, void *v);
extern int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
```

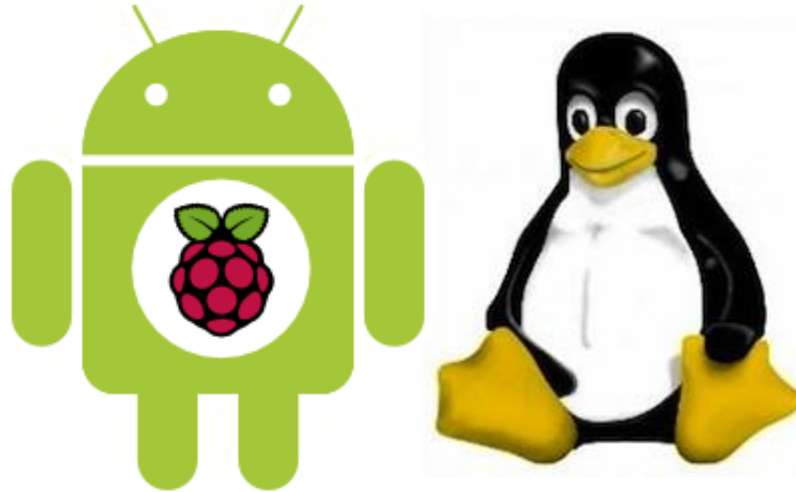

이 페이지는 의도적으로 비워둔 것임.

TODO

- 1) Beagle Board(<http://beagleboard.org/>)
- 2) Raspberry Pi(<http://www.raspberrypi.org/>)

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) *코드로 알아보는 ARM Linux Kernel* [노서영 외, Jpub]
- 6) *Linux Kernel architecture for device drivers ...* [Thomas Petazzoni Free Electronics
(thomas.petazzoni@free-electronics.com)]
- 7) *The sysfs Filesystem* [Patrick Mochel, mochel@digitalimplant.org]
- 8) *Linux SD/MMC Driver Stack* [Champ Yen, champ.yen@gmail.com]
- 9) *%233.GTUG-Android-Power Management.pdf ...* [Renaldo Noma 2010]
- 10) *Android_Debug_Guide6.pdf* [Chunghan Yi]
- 11) *안드로이드 아나토미 시스템 서비스* [김태연/박지훈/김상엽/이왕재, 개발자가 행복한 세상]
- 12) *InterruptThreads-Slides_Anderson.ppt* [Mike Anderson, mike@theptrgroup.com]
- 13) *Interrupt handling* [Andrew N. Sloss (asloss@arm.com)]
- 14) *Some Internet Articles ...*



SlowBoot