Android Kernel Hacks

안드로이드를 위한 리눅스 커널 핵스

2013.7.16 ~ 2013.9.27

이 충 한(chunghan.yi@gmail.com, slowboot)

목차

CHAPTER 2

주요 Kernel 프로그래밍 기법 1

본 장에서는 안드로이드의 핵심인 리눅스 커널을 소개하고, 커널 프로그래머가 놓치기 쉬운 커널 프로그래밍 기법에 대하여 소개하고자 한다.

- Kernel 초기화 과정 분석

- Task, Wait Queue, Scheduling, 그리고 Preemption
- Top Half, Bottom Halves와 Deferring Work
  - Hard Interrupt Handler, Softirq, Tasklet, Work Queue, Interrupt Thread
- Timer

- 동기화 방식(Synchronization Method)
  - Spinlock, Mutex, Semaphore, Completion, RCU
- Notifier Chains
- 메모리 관련 프로그래밍 기법
- Device Model & Sysfs

커널 프로그래밍을 제대로 하려면 과연 어떻게 해야 할까? 본서에서는 이 질문에 대한 대답으로 기존의 커널 프로그래밍 관련 서적에서 다루고 있는 방식과는 다른 보다 실질적인 커널 프로그래밍 기법을 위주로 설명을 진행해 보고자 한다. 이번 장에서 설명하려는 내용을 요약해 보면 다음과 같다.

1) Task scheduling과 Wait Queue 이해
2) Interrupt 원리 및 interrupt handler 구현 방법
3) Tasklet, SoftIRQ 보다는 최근 많은 사용량을 보이는 Work Queue 중심으로 Bottom Half 설명
4) Kernel thread 작성 방법과 Interrupt Thread 소개
5) 주요 Synchronization 기법 소개 – 그 중에서도 Completion, RCU 위주의 설명
6) Notifier chain에 근간한 event 전달 방법 소개
7) Memory 관련 여러 기법 소개
8) Device Model과 Sysfs 소개

1. 커널 초기화 과정 분석

커널 초기화 과정은 그 자체가 하나의 책으로 정리해도 모자랄 만큼 매우 방대한 영역이다[참고문헌 5참조]. 따라서 본 절에서는 이렇게 복잡한 커널 초기화 과정을 세세하게 정리하는 것 보다는 간략하게 핵심만을 요약해 봄으로써 커널 초기화의 전반적인 개념을 파악해 보고자 한다.

부트로더로부터 init process가 실행되기까지의 과정을 요약해 보면 다음 그림과 같다.

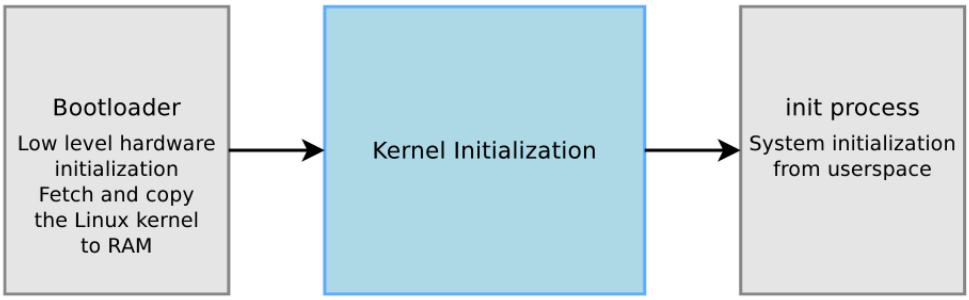

그림 2-1 커널 초기화 과정 [출처 – 참고 문헌 8]

Kernel Bootstrap

커널 빌드 시, kernel bootstrap이 어떤 식으로 표시되는지를 살펴 보기로 한다. 아래 예는 ARM (pxa cpu)을 target으로 하는 Linux 2.6.36을 빌드하는 것을 보여주고 있다.

```
...
LD vmlinux
SYSMAP System.map
SYSMAP .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS arch/arm/boot/compressed/head.o
GZIP arch/arm/boot/compressed/piggy.gzip
AS arch/arm/boot/compressed/piggy.gzip.o
CC arch/arm/boot/compressed/misc.o
CC arch/arm/boot/compressed/decompress.o
AS arch/arm/boot/compressed/head-xscale.o
SHIPPED arch/arm/boot/compressed/lib1funcs.S
AS arch/arm/boot/compressed/lib1funcs.o
LD arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
...
```

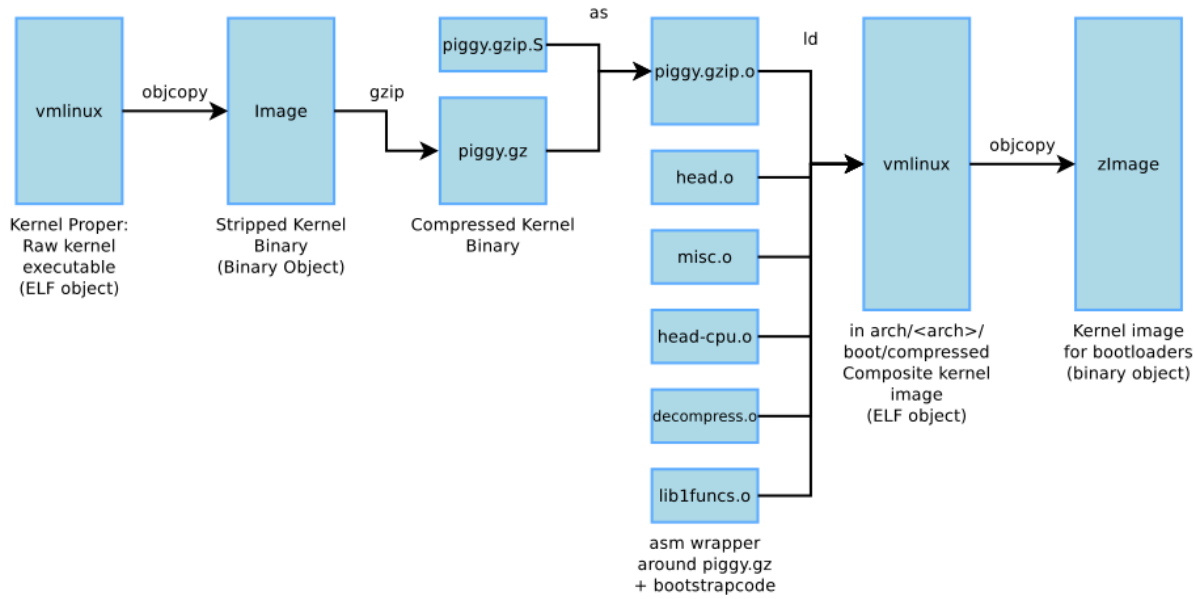

그림 2-2 커널 빌드 시 이미지 생성 과정 [출처 – 참고 문헌 8]

압축된 커널을 위한 bootstrap 코드는 arch/<arch>/boot/compressed 디렉토리에 위치하며, 커널 스스로 압축을 풀고, 시작 지점으로 jump하도록 해준다.

1) head.o는 아키텍쳐에 특화(Architecture specific)된 초기화 코드이며, bootloader가 실행한다.

2) head-cpu.o (여기서는 head-xscale.o) CPU에 특화(CPU specific) 된 초기화 코드이다.

3) decompress.o, misc.o는 압축을 풀어주는 것과 관련된 코드이다.

4) piggy.<compressionformat>.o이 바로 커널 자신이다.

아키텍쳐 특화(architecture-specific)된 초기화 코드

커널이 스스로 압축을 푼 후, 커널의 시작 부분(kernel entry point)로 분기하게 되는데, 이를 담고 있는 파일은 arch/<arch>/kernel/head.S 이며, 주요 임무는 다음과 같다.

1) architecture, 프로세서, 머신 유형 등을 검사한다.

Check the architecture, processor and machine type.

2) MMU를 설정하고, page table 항목을 만든 후, 가상 메모리(virtual memory)를 enable 시킨다.

3) init/main.c 파일에 있는 start_kernel() 함수를 호출한다.

start kernel 함수에 주로 하는 일

1) setup_arch(&command_line) 함수를 호출해 준다.
   - arch/<arch>/kernel/setup.c 파일에 정의되어 있는 함수이다.
   - Bootloader가 넘긴 command line 값을 복사한다.
   - ARM에서는 이 함수는 setup_processor() 함수(CPU 정보가 출력됨)와 setup_machine() 함수(머신을 초기화 시켜줌)를 다시 호출한다
2) 에러 메시지를 출력할 수 있도록, 가능한 한 일찍 콘솔을 초기화해 준다.
3) 다양한 커널 subsystem을 초기화 시켜준다.
4) 최종적으로 rest_init() 함수를 호출한다.

rest init: init process 시작하기

```
static noinline void __init_refok rest_init(void)
{
	int pid;
	rcu_scheduler_starting();
	/*
	* We need to spawn init first so that it obtains pid 1, however
	* the init task will end up wanting to create kthreads, which, if
	* we schedule it before we create kthreadd, will OOPS.
	*/
	kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
	numa_default_policy();
	pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
	rcu_read_lock();
	kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
```

```
        rcu_read_unlock();
        complete(&kthreadd_done);
        /*
        * The boot idle thread must execute schedule()
        * at least once to get things moving:
        */
        init_idle_bootup_task(current);
        preempt_enable_no_resched();
        schedule();
        preempt_disable();
        /* Call into cpu_idle with preempt disabled */
        cpu_idle();
}
```

kernel init

kernel_init 함수는 두 가지 중요한 일을 수행한다.

1) do_basic_setup () 함수를 호출한다(kernel_init_freeable() 함수 내에서 호출함).

```
static void __init do_basic_setup(void)
{
        cpuset_init_smp();
        usermodehelper_init();
        init_tmpfs();
        driver_init();
        init_irq_proc();
        do_ctors();
        do_initcalls();
}
```

2) init_post() 함수를 호출한다. 참고로, 본서에서 참조하고 있는 kernel 3.10.3에서는 init_post() 함수는 없어지고, 이에 상응하는 코드(console open 및 init process 실행)만이 존재하는 형태로 구조가 좀 바뀌었다. 하지만 내용 전개상 커다란 문제가 되지 않을 듯하여, init_post()가 있는 것으로 보고 설명을 진행하고자 한다.

do initcalls

do_initcalls() 함수에서 하는 일은 아래에 열거되어 있는 매크로를 사용하여 기 등록한 hooking 함수(pluggable hooks – 사용자 등록한 모듈 등이 여기에 해당함)를 호출해 주는 것이다. 이 방식이 갖는 장점은 커널이 사용자가 추가한 모든 드라이버 등을 알고 있을 필요가 없다는 점이다.

```
From defined in include/linux/init.h
```

```
/*
* A "pure" initcall has no dependencies on anything else, and purely
* initializes variables that couldn't be statically initialized.
*
* This only exists for built-in code, not for modules.
*/
#define pure_initcall(fn) __define_initcall("0",fn,1)
#define core_initcall(fn) __define_initcall("1",fn,1)
#define core_initcall_sync(fn) __define_initcall("1s",fn,1s)
#define postcore_initcall(fn) __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn) __define_initcall("2s",fn,2s)
#define arch_initcall(fn) __define_initcall("3",fn,3)
#define arch_initcall_sync(fn) __define_initcall("3s",fn,3s)
#define subsys_initcall(fn) __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn) __define_initcall("4s",fn,4s)
#define fs_initcall(fn) __define_initcall("5",fn,5)
#define fs_initcall_sync(fn) __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn) __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn) __define_initcall("6",fn,6)
#define device_initcall_sync(fn) __define_initcall("6s",fn,6s)
#define late_initcall(fn) __define_initcall("7",fn,7)
#define late_initcall_sync(fn) __define_initcall("7s",fn,7s)
```

위에서 언급한 initcall을 사용하는 예를 아래에 들어 보았다. device_initcall()의 경우는 우선순위가 6으로 되어 있으므로, 나중에 실행됨을 알 수 있다.

From arch/arm/mach-pxa/lpd270.c (Linux 2.6.36)

```
static int __init lpd270_irq_device_init(void)
{
	int ret = -ENODEV;
	if (machine_is_logicpd_pxa270()) {
	ret = sysdev_class_register(&lpd270_irq_sysclass);
	if (ret == 0)
		ret = sysdev_register(&lpd270_irq_device);
	}
	return ret;
}
device_initcall(lpd270_irq_device_init);
```

앞으로 3장에서 설명할 SoC/보드 초기화 부분은 실제로 이 단계(do_initcalls() 함수)를 수행하는 도중에 호출된다. 아래 그림 2-3을 참조하기 바란다.

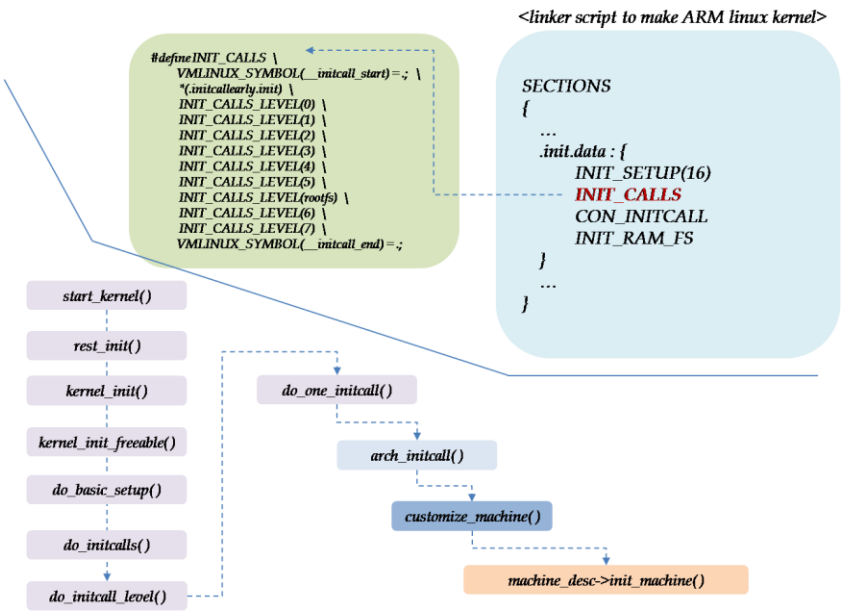

그림 2-3 머신 코드(SoC/보드 초기화 코드) 실행

init post

커널 부팅의 마지막 단계로, console을 오픈하려고 하며, init process를 실행시킨다.

```
static noinline int init_post(void) __releases(kernel_lock) {
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();
    current->signal->flags |= SIGNAL_UNKILLABLE;
    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s₩n", ramdisk_execute_command);
    }
    /* We try each of these until one succeeds.
    * The Bourne shell can be used instead of init if we are
    * trying to recover a really broken machine. */
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s. Attempting defaults...₩n",
```

```
                        execute_command);
        }
        run_init_process("/sbin/init");
        run_init_process("/etc/init");
        run_init_process("/bin/init");
        run_init_process("/bin/sh");
        panic("No init found. Try passing init= option to kernel. See Linux Documentation/init.txt");
}
```

커널 초기화 과정을 요약해 보면 다음과 같다.

1) bootloader는 bootstrap code를 실행시킨다.
2) Bootstrap 코드는 프로세서와 보드를 초기화하고, 커널의 압축을 풀어 RAM에 적재한 후, start_kernel() 함수를 호출해 준다.
3) 커널은 bootloader로부터 command line option을 복사해 온다.
4) 커널은 프로세서와 머신을 초기화시킨다.
5) 콘솔을 초기화한다.
6) 메모리 할당(memory allocation), scheduling, 파일 cache 등 커널 서비스를 초기화 시킨다.
7) 커널 thread(나중에 init process)를 생성하고, idle loop 상태에서 대기한다.
8) 장치를 초기화하고, initcall 매크로를 호출한다.

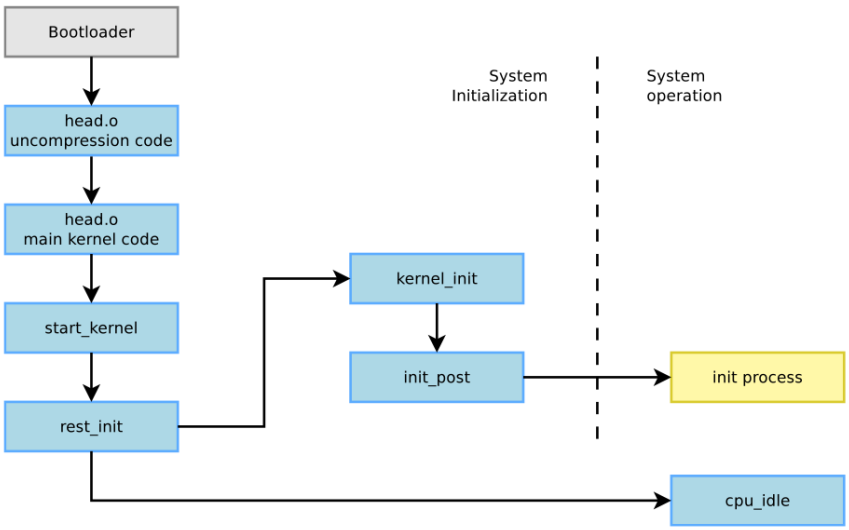

그림 2-4 커널 초기화 과정 [출처 – 참고 문헌 8]

마지막으로, 위에서는 자세히 언급하지는 않았으나, CPU가 2개 이상인 경우 부팅 과정을 간략히 그림으로 표현해 보았으니, 아래 그림 2-5를 참조하기 바란다.

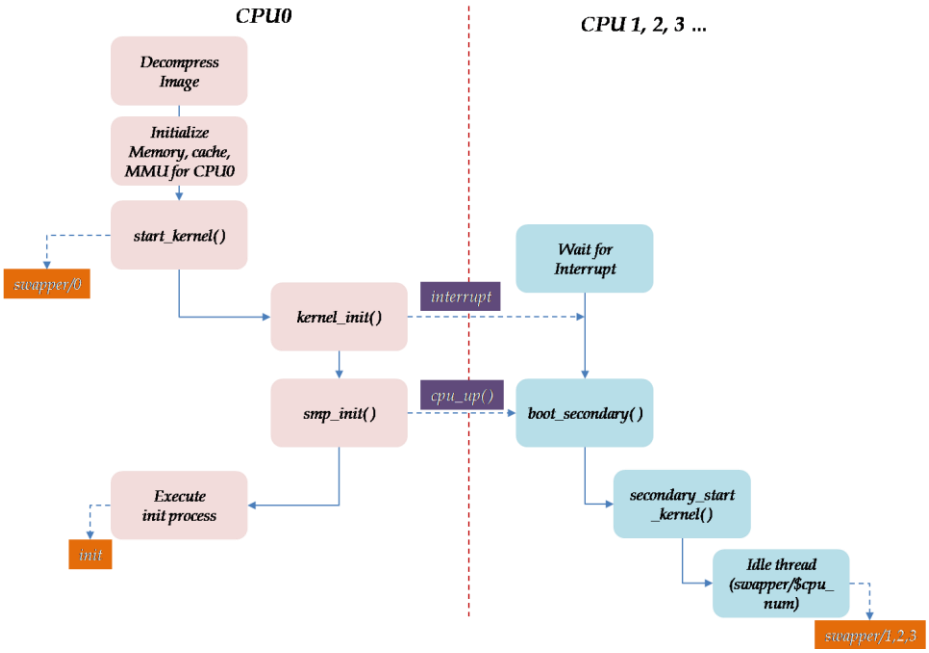

그림 2-5 SMP kernel booting 순서

2. 타스크 스케줄링과 우선순위 선점

타스크 스케쥴링(Task Scheduling)

Linux에서는 기본적인 실행 단위인 프로세스를 위한 각종 정보를 담기 위해, task_struct 구조체가 사용된다. task_struct는 프로세스와 관련된 다양한 자원 정보를 저장하고, 커널 코드를 실행하기 위한 스택과 저 수준의 플래그(flag)는 thread_info structure에 저장된다. 참고로, task_struct data structure는 32bit CPU 기준으로 약 1.7KB의 크기(매우 큼)를 필요로 한다.

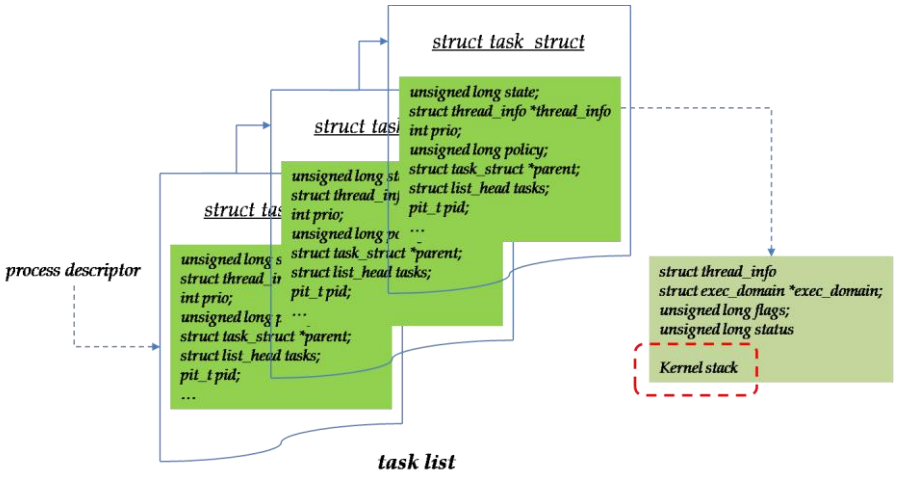

그림 2-5 타스크 스케쥴링(2) – task list

task 관련 queue로는 wait queue와 run queue가 있으며, run queue에 등록된 task는 실행되고, wait queue에 등록된 task는 특정 조건이 성립될 때까지 기다리게 된다.

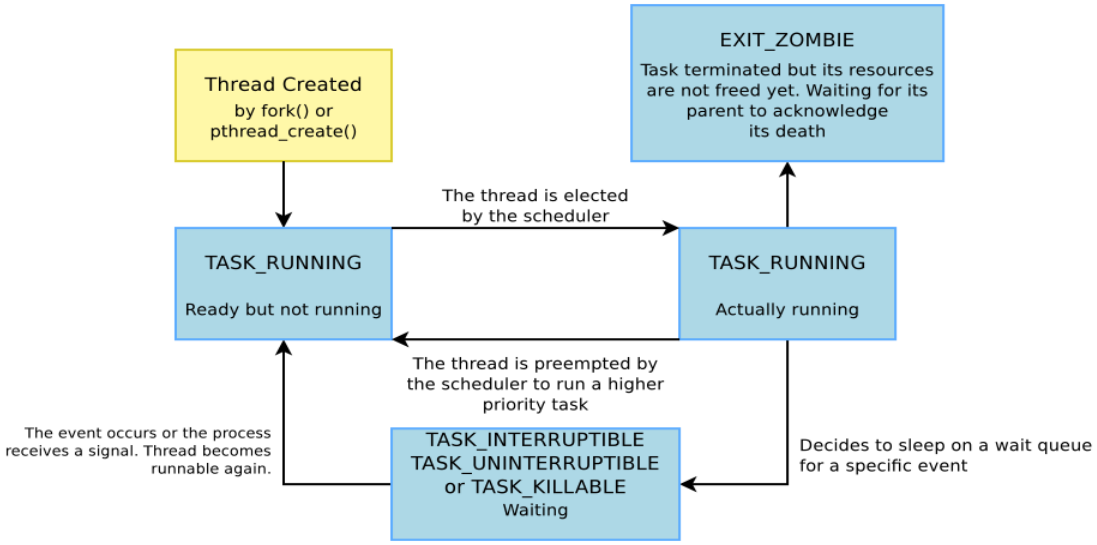

그림 2-6 타스크 스케쥴링(3) – run queue/wait queue [출처 – 참고 문헌 8]

그림 2-7은 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 내에서 명시적으로 수행하는 경우도 많다.

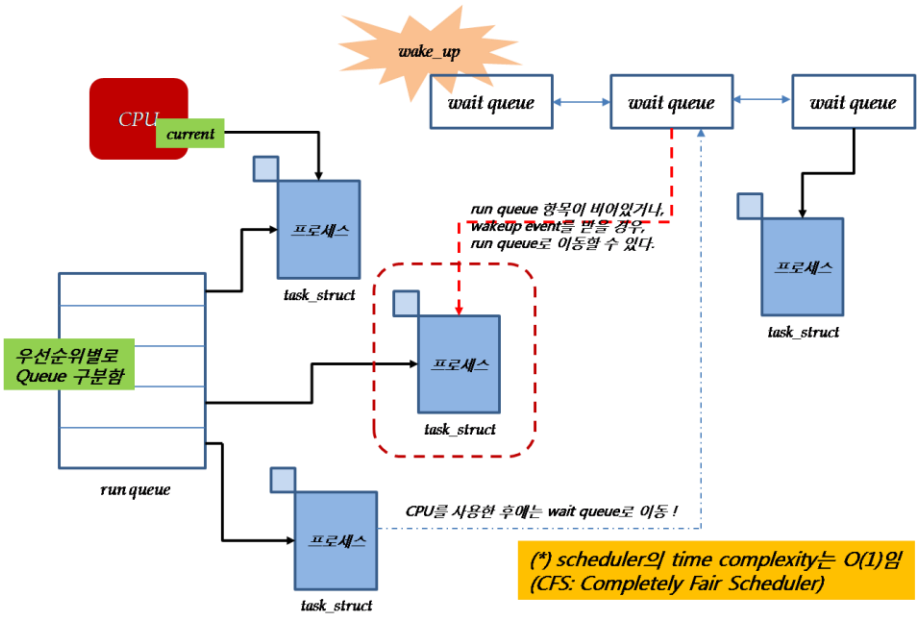

그림 2-7 타스크 스케쥴링(1)

지금부터는 task(kernel space 혹은 user space process)가 데이터를 기다려야 할 때 발생하는 sleeping 상황(scheduling 상황)에 대해 좀 더 살펴보기로 하자.

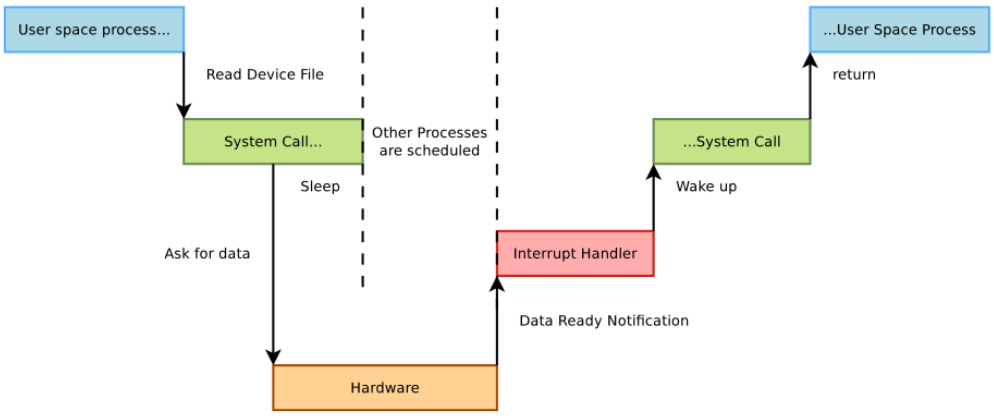

그림 2-8 Sleeping 개념 [출처 – 참고 문헌 8]

그림 2-8은 user space의 프로세스가 read() system call을 사용하여 device(hardware)로부터 데이터를 읽어 들이는 과정을 묘사하고 있다. 이 과정을 요약해 보면 다음과 같다.

1) read() 함수(system call)를 호출한 user process는 device(hardware)로부터 데이터가 준비될 때까지 sleep 상태(wait queue에 들어감)에 빠진다 – wait_event() 함수를 명시적으로 호출함.

2) 이 기간 동안에는 다른 task가 CPU를 점유하도록 scheduling이 이루어지게 된다.

3) 마침내 device로부터 데이터가 준비가 된 경우, device는 hardware interrupt를 발생시키게 된다.

4) device driver의 interrupt handler 함수 안에서는 wake_up() 함수를 호출하여, sleep 상태에 있던 task를 깨워 준다.

5) 잠에서 깨어난 task는 실제 데이터를 읽어 들인 후, read() system call을 종료한다.

이상의 과정에서 사용한 wait_event() 및 wake_up() 함수에 대해 좀 더 자세히 정리해 보기로 하자.

[Wait Queue 관련 주요 API 소개 – include/linux/wait.h 참조]

<변수 선언 및 초기화>

wait_queue_head_t wq;

***init_waitqueue_head**(&wq);*

혹은

***DECLARE_WAIT_QUEUE_HEAD**(wq);*

<Sleep 상태로 들어감>

***wait_event**(wq, condition)*

- 매크로임
- condition 값이 true가 될 때가지 sleep 함((TASK_UNINTERRUPTIBLE).

wait_event_interruptible*(wq, condition)*

- 매크로임
- condition 값이 true가 될 때가지 sleep 함(TASK_INTERRUPTIBLE).

wait_event_killable*(wq, condition)*

- 매크로임
- condition 값이 true가 될 때가지 sleep 함(TASK_KILLABLE – SIGKILL에 의해서만 인터럽트 될 수 있음을 의미).

wait_event_timeout*(wq, condition, timeout)*

- 매크로임
- condition 값이 true가 되거나, timeout이 경과할 때까지 slee 함 (TASK_UNINTERRUPTIBLE).

…

<Sleep에서 깨어남>

#define ***wake_up****(x)* *__wake_up(x, TASK_NORMAL, 1, NULL)*

- 매크로임
- Waitqueue에서 대기 중인 kernel thread를 깨워 줌(TASK_NORMAL).

#define ***wake_up_interruptible****(x)* *__wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)*

- 매크로임
- Waitqueue에서 대기 중인 kernel thread를 깨워 줌(TASK_INTERRUPTIBLE).

아래 코드는 wait queue에서 대기 중이던 task를 꺼내어 CPU를 할당해 주는 부분으로, 실제로 wait_event() 함수의 내부를 구성하고 있다. 이와 같은 스타일의 코드가 실제로 커널 코드 이곳 저곳에서 매우 많이 사용되므로 눈 여겨 보기 바란다.

```
#define __wait_event(wq, condition)                                    ₩
do {                                                                   ₩
    DEFINE_WAIT(__wait); ); ①                                          ₩
    for (;;) {                                                         ₩
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); ②         ₩
        if (condition) ③                                               ₩
            break;                                                     ₩
```

```
        schedule(); ④                                    ₩
    }                                                    ₩
    finish_wait(&wq, &__wait); ⑤                         ₩
} while (0)
```

① 매크로를 이용하여 wait queue entry를 하나 생성한다.

② 자신을 wait queue에 넣고, task 상태를 TASK_INTERRUPTIBLE로 바꾼다. Wait queue는 깨어날 조건이 발생할 때, 해당 task를 깨운다.

③ wakeup 조건이 성립(wake_up() 함수 호출)되면, for loop을 빠져 나온다. 이제 task는 깨어날 수 있는 상태가 된다.

④ wakeup할 조건이 성립되지 않으면, 해당 task는 여전히 sleep한다. 즉, 다른 task가 CPU를 사용할 수 있도록 schedule() 함수를 호출해 준다.

⑤ task 상태를 TASK_RUNNING으로 바꾸고, 깨어 나기 위해 자신을 wait queue에서 제거한다

[참고 사항] 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.

[예제 코드 - drivers/staging/android/binder.c]

```
static DECLARE_WAIT_QUEUE_HEAD(binder_user_error_wait);   //wait queue 선언    ①
static int binder_stop_on_user_error;
[...]
static int binder_set_stop_on_user_error(const char *val, struct kernel_param *kp)
{
    int ret;
    ret = param_set_int(val, kp);
    if (binder_stop_on_user_error < 2)
        wake_up(&binder_user_error_wait);    // 대기 중인 kernel thread를 깨워 줌    ④
    return ret;
}
module_param_call(stop_on_user_error, binder_set_stop_on_user_error,
                  param_get_int, &binder_stop_on_user_error, S_IWUSR | S_IRUGO);
```

```
static int binder_thread_read(struct binder_proc *proc,      //binder thread read 함수    ②
                              struct binder_thread *thread,
                              void  __user *buffer, int size,
                              signed long *consumed, int non_block)
{
    [...]
    if (wait_for_proc_work) {
        if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
                                BINDER_LOOPER_STATE_ENTERED))) {
            binder_user_error("%d:%d ERROR: Thread waiting for process work before calling ₩
                              BC_REGISTER_LOOPER or BC_ENTER_LOOPER (state %x)₩n",
                              proc->pid, thread->pid, thread->looper);
            wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
            // condition 값이 true가 될 때까지 sleep 함    ③
        }
        binder_set_nice(proc->default_priority);
        if (non_block) {
            if (!binder_has_proc_work(proc, thread))
                ret = -EAGAIN;
        } else
            ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work(proc, thread));
    }
    [...]
}
```

스케쥴링 클래스와 우선 순위

Runqueue에 존재하는 task는 scheduler class 별로 분류가 되는데, 이 scheduler class 별로 우선 순위가 구분되어 있다. Real-time scheduling policy(SCHED_FIFO, SCHED_RR)로 지정된 task는 CFS(Completely Fair Scheduler) policy(SCHED_NORMAL)로 지정된 task 보다 무조건 먼저 schedule된다. 참고로, 그림 2-9에서 볼 수 있는 것처럼, SCHED_FIFO나 SCHED_RR은 0 ~ 99의 우선 순위 값을 가지며, SCHED_NORMA과 SCHED_BATCH는 100 ~ 139의 우선 순위 값을 갖는다. 작은 수가 높은 우선 순위임을 감안할 때, Real-Time Scheduling Class로 할당된 task가 그렇지 않은 task에 비해 항상 먼저 scheduling된다고 보면 된다.

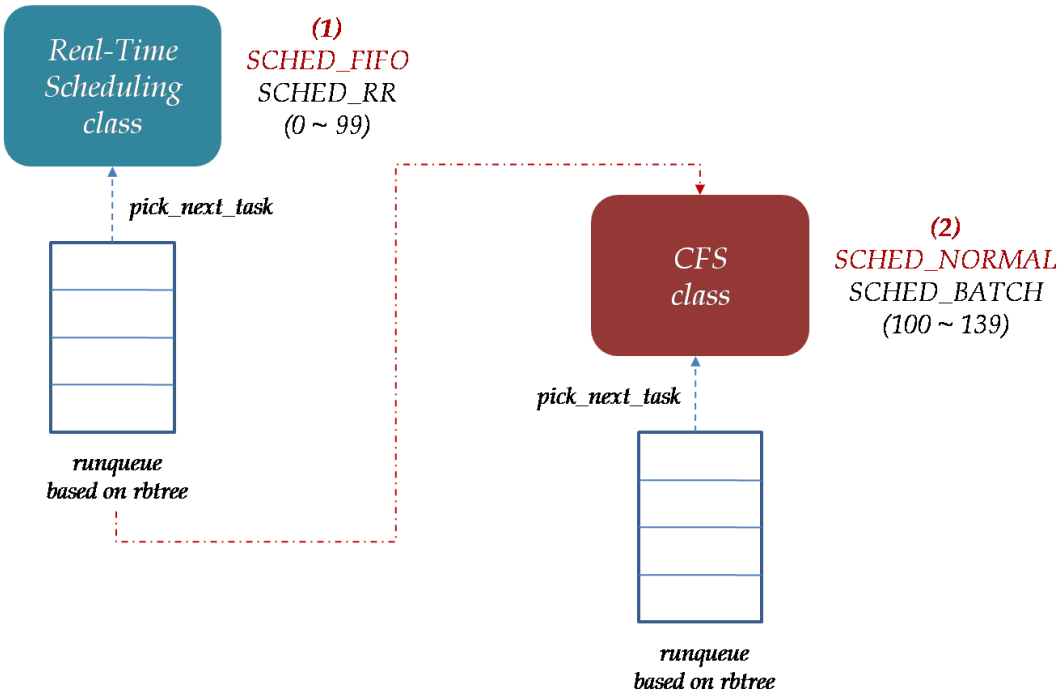

그림 2-9 타스크 스케쥴링(4) – scheduling class와 우선 순위

우선순위 선점

Linux kernel 2.6 이상은 interrupt, system call, 다중 프로세서 등의 다양한 상황에서 **fully preemptive**하다. 즉, 임의의 task가 현재 실행중인 task로부터 CPU를 빼앗을 수 있다. preemption되면 runnable task가 다른 task로 교체 (context switching – 그림 2-11 참조)된다. 즉, schedule() 함수가 호출된다. 아래 그림 2-10에서 user Preemption 및 kernel Preemption 발생 시점을 정리하면 다음과 같다.

<User preemption 시점>

1) System call 후, user space로 복귀 시

2) Interrupt handler 처리 후, user space로 복귀 시

<Kernel preemption 시점>

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)

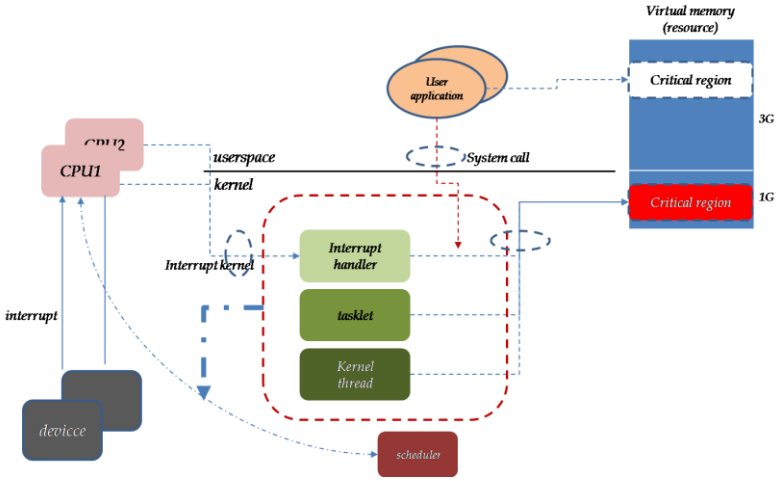

그림 2-10 Kernel Preemption

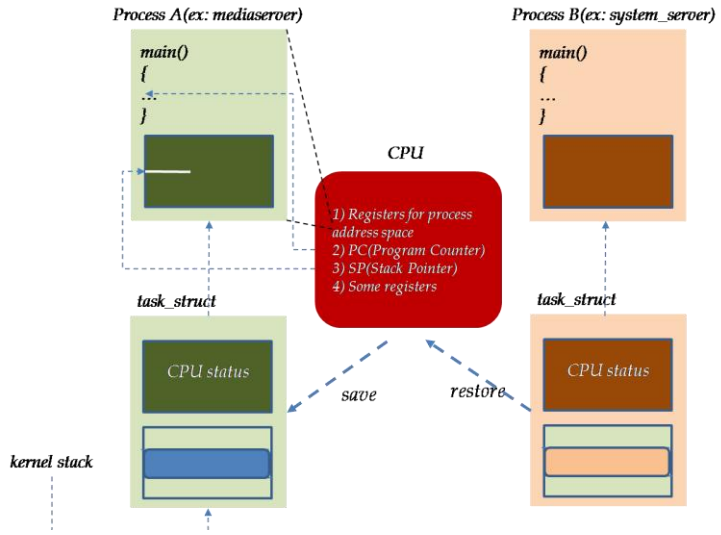

그림 2-11 Context Switching 개념

3. Top Half와 Bottom Half

Linux kernel에서 말하는 Top Half와 Bottom Half의 개념을 정리하면 다음과 같다.

1) Interrupt handler를 top half라고 하며, 지연 처리(deferring work)가 가능한 루틴을 bottom half 라고 함.

- *지연 처리는 interrupt context(softirq, tasklet)에 대해서도 필요하며, process context(work queue)에도 필요하다.*

2) Bottom half 중에서도 해당 작업이 sleep 가능하거나 sleep이 필요할 경우, ***work queue****를 사용*

한다.

- *process context에서 실행*

3) 2의 조건에 해당하지 않으며 빠른 처리가 필수적인 경우, ***tasklet****를 사용한다.*

- *interrupt context에서 실행*
- *Softirq도 tasklet과 동일한 구조이나, 사용되는 내용이 정적으로 정해져 있음. 따라서 programming 시에는 동적인 등록이 가능한 tasklet이 사용된다고 이해하면 될 듯함.*

4) tasklet과 softirq의 관계와 마찬가지로, work queue는 kernel thread를 기반으로 하여 구현되어 있다.

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 듯하다.*

Bottom Half의 개념

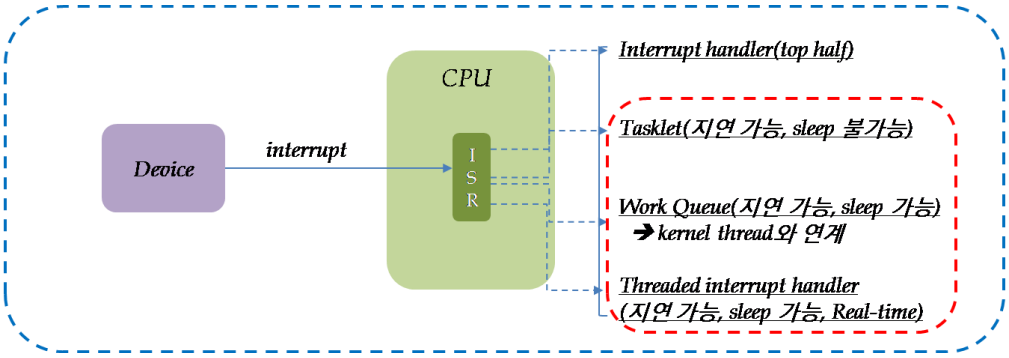

그림 2-12 Top Half와 Bottom Half의 개념(1)

위의 그림에서처럼, 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 지를 결정해야 한다. work queue는 bottom half 개념으로 등장하기는 했으나, 위에서와 같이 interrupt handler 내에서의 지연 처리뿐만 아니라, 임의의 process context에 대한 지연 처리 시에도 널리 활용되고 있다.

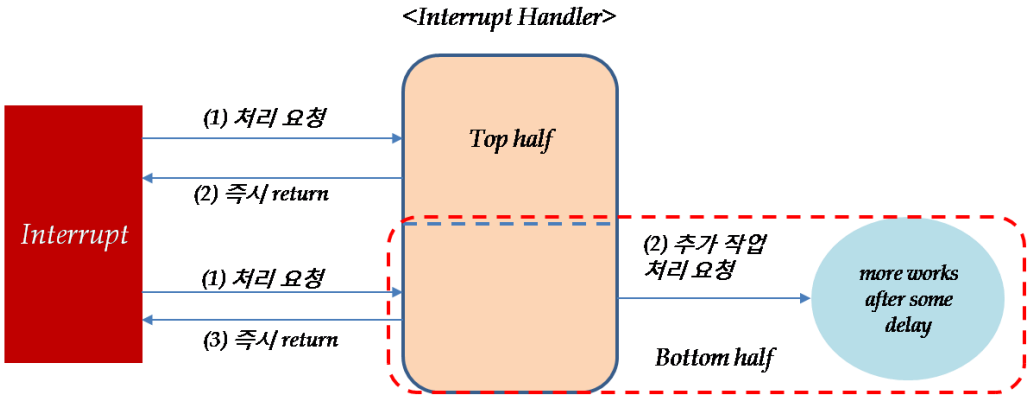

그림 2-13 Top Half와 Bottom Half의 개념(2)

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 리턴 하므로써, 다음 interrupt의 유실을 최대한 막을 수 있는 것으로 이해하면 될 듯하다.

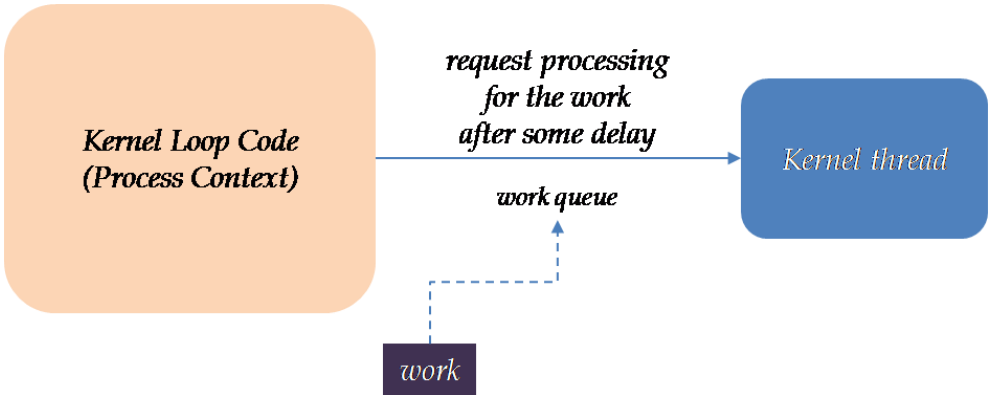

그림 2-14 프로세스 컨텍스트와 워크 큐

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 사용된다. 따라서 앞서 이미 언급한 바와 같이, work queue의 경우는 interrupt handler 내에서의 지연 처리뿐만 아니라, 임의의 process context에 대한 지연 처리에도 널리 활용되고 있다.

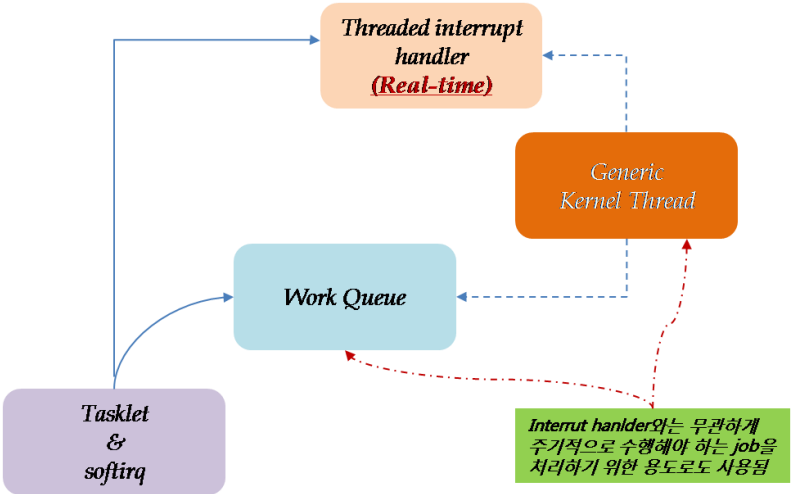

그림 2-15 Bottom Havles 모음

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

Interrupt Context와 Process Context

User application에 의해 발생하는 system call을 처리하는 kernel code의 경우 process context에서 실행된다고 말한다. process context에서 실행되는 kernel code는 다른 kernel code에 의해 CPU 사용을 빼앗길 수 있다(preemptive). Process context의 대상으로는 user process로 부터 온 system call 처리, work queue, kernel thread, threaded interrupt handler 등이 있다. 반면에 interrupt handler를 interrupt context라고 이해하면 쉬울 듯하다. interrupt context에서 수행되는 kernel code는 끝날 때까지 다른 kernel code에 의해 중단될 수 없다. interrupt context의 대상으로는 hard interrupt handler, softirq/tasklet 등이 있다.

<인터럽트 컨텍스트의 제약 사항>

1) Sleep하거나, processor를 포기

2) Mutex 사용

3) 시간을 많이 잡아 먹는 일

4) User space(virtual memory) 접근

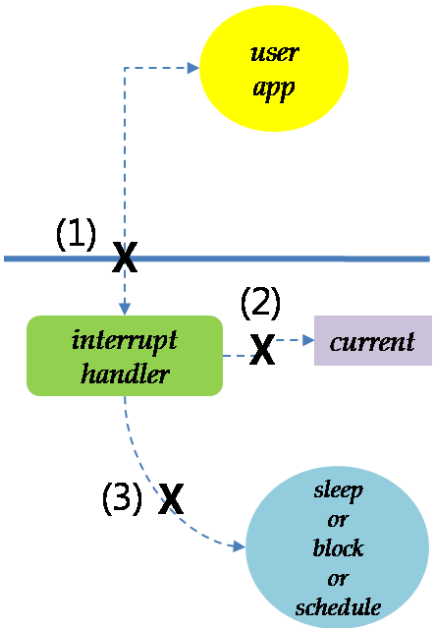

그림 2-16 인터럽트 컨텍스트의 제약 사항

인터럽트 컨텍스트의 제약 사항을 좀 더 구체적으로 정리해 보면 다음과 같다.

1) 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)

2) 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).

3) Sleeping이 불가하며, scheduling도 할 수 없다. Atomic code는 schedule()함수를 call해서는 안 되며, wait_event나 sleep으로 갈 수 있는 어떠한 형태의 함수를 호출해서도 안 된다. 예를 들어, kmalloc(..., GFP_KERNEL) 을 호출해서는 안 된다(GFP_ATOMIC을 사용해야 함). Semaphore도 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 적용된다. 반대로, work queue는 process context에서 동작하므로 위의 제약 사항을 모두 사용할 수 있다.

3.1 인터럽트 핸들러(Top Half)

Interrupt Handler 내부 처리 구조

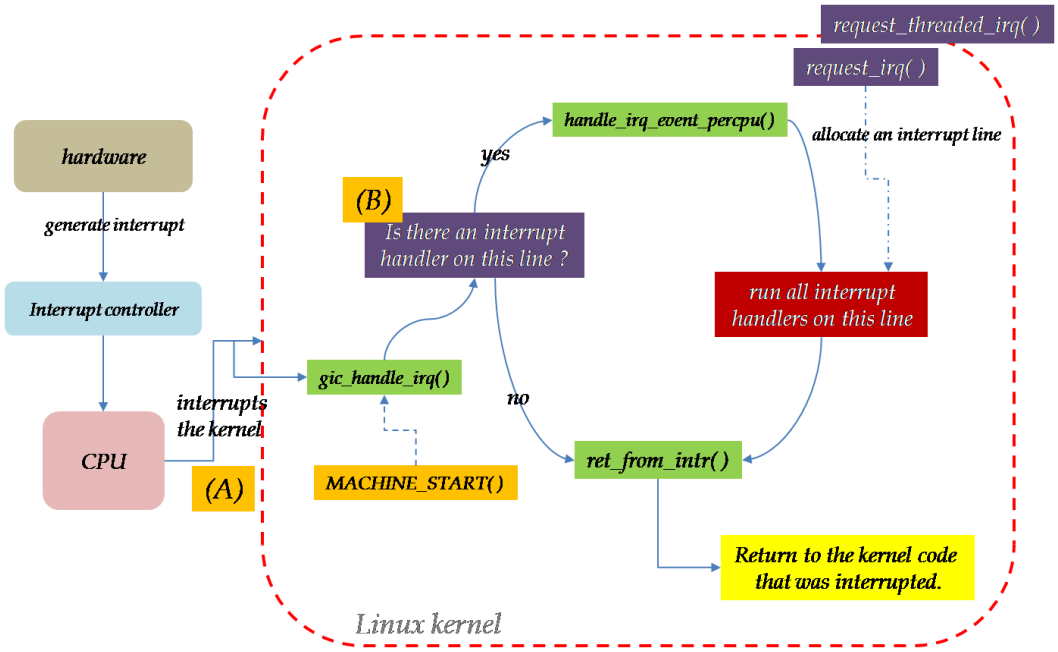

그림 2-17 인터럽트 핸들러 동작 방식 이해

인터럽트 및 인터럽트 핸들러가 동작하는 방식을 정리하면 다음과 같다.

1) 사용자(kernel programmer)는 request_irq( ), request_threaded_irq( ) 등의 함수를 사용하여 특정 H/W 장치에 대한 interrupt handler를 등록하는 코드(디바이스 드라이버)를 작성해 둔다.

2) H/W 장치로부터 인터럽트가 발생하면, interrupt controller를 통해 CPU로 interrupt가 전달된다.

3) Interrupt를 받은 CPU는 하던 일을 멈추고, interrupt 처리 루틴으로 전환한다.

- *CPSR의 모드를 IRQ mode로 변경*
- *PC 값, CPSR 값을 r0 register에 저장*
- *IRQ가 disable된 상태에서 ARM mode로 변경*
- *User mode exception -> __irq_usr, kernel mode exception -> __irq_svc로 jump*

4) arch/arm/kernel/entry-armv.S 코드 내에 CONFIG_MULTI_IRQ_HANDLER=y(chip 제조사가 정의한 interrupt handler 사용 의미) 가 정의되어 있지 않은 경우, linux default IRQ handler 처리 루틴인 asm_do_IRQ( ) 함수로 분기하고, 그렇지 않을 경우 handle_arch_irq( ) 함수로 분기한다.

- *handle_arch_irq( ) = handle_irq( ) = gic_handle_irq( )*
- *본 서에서는 handle_arch_irq( )의 흐름을 살펴볼 것임(Qualcomm codes)*

5) gic_handle_irq( ) 함수는 아래의 함수 흐름을 따라 최종적으로 handle_irq_event_percpu( ) 함수를 호출하게 되며, 이 함수 내에서 사용하자 이미 등록해 둔 인터럽트 핸들러를 수행한다.

- *handle_IRQ( ) -> generic_handle_irq( ) -> handle_fasteoi_irq( ) -> handle_irq_event( ) -> handle_irq_event_percpu( )*

- *그림 2-19 참조*

6) handle_irq_event_percpu( ) 함수에서 수행하는 인터럽트 핸들러는 1단계에서 사용자가 등록한 방식에 따라 크게 두가지로 분류하여 실행된다.

- *request_irq( ) 의 경우 : action->handler( ) 함수 호출*
- *request_threaded_irq( ) 의 경우: irq_wake_thread( ) 함수 호출. 잠자던 thread를 깨어 실행시켜 줌.*

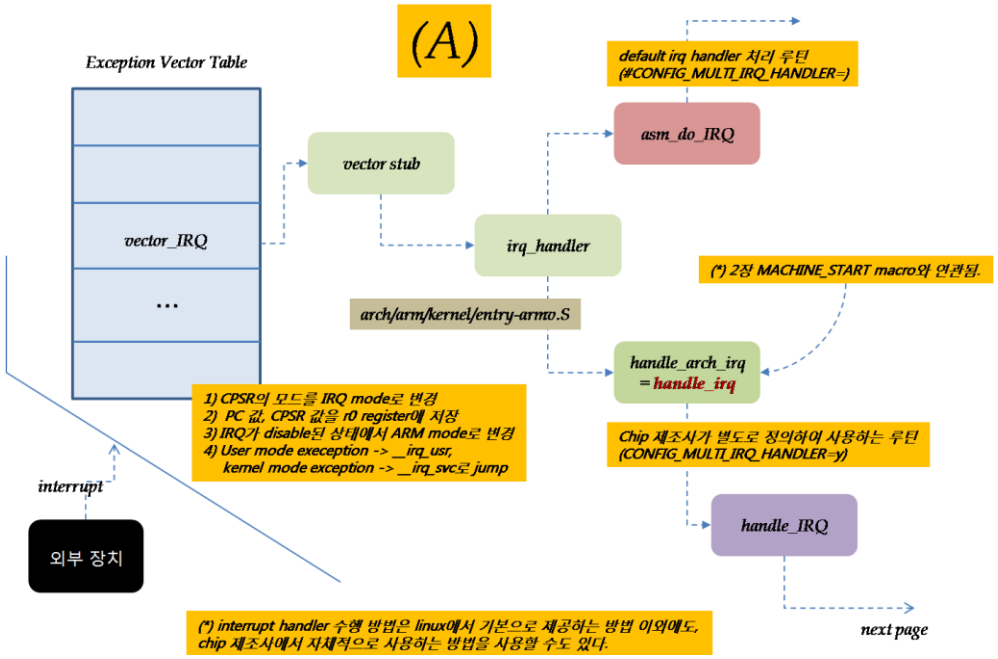

그림 2-18 Linux default & 제조사 정의 IRQ handle code

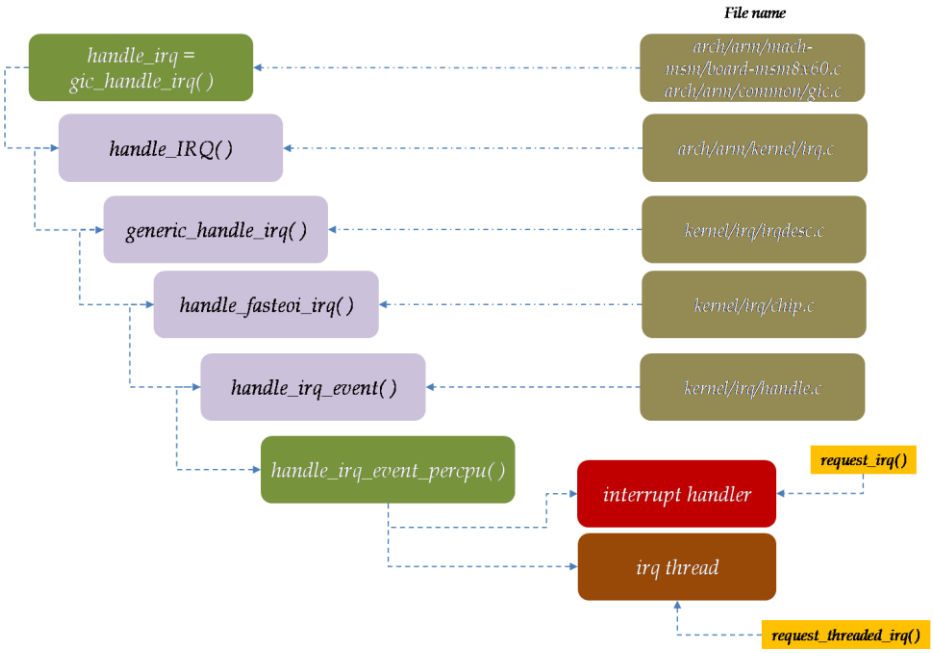

그림 2-19 handle_irq_event_percpu( ) 함수 호출 경로

인터럽트 핸들러를 최종적으로 수행해 주는 handle_irq_event_percpu( ) 함수의 주요 부분만을 발췌해 보면 다음과 같다.

kernel/irq/handle.c에서 발췌

```
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)
{
        ...
      do {
            irqreturn_t res;

            res = action->handler(...);       /* request_irq()로 등록한 irq handler 실행 */
            WARN_ONCE(!irqs_disabled(), ....);
            /* local irq가 disable되어 있지 않으면, warning message 출력 */
            /* interrupt 처리 중에는 다른 interrupt가 들어오면 안됨. */

            switch (res) {
            case IRQ_WAKE_THREAD:             // interrupt thread 이면 – 9장 관련 사항임.
                irq_wake_thread(irq, action)

            case IRQ_HANDLED:
                flags |= action->flags;

            default:
                 break;
            }
            action = action->next;          /* 다음 action 선택 */
                                            /* 같은 interrupt line, 복수개의 handler 등록 시 사용 */
      } while (action);
      ...
}
```

Interrupt Handling 관련 주요 API 소개

앞서 설명한 바와 같이 커널 프로그래머는 request_irq() 함수를 사용하여 H/W 장치로부터 interrupt 발생 시, 처리할 함수를 등록해 주게 된다. 이 함수를 interrupt handler라고 하며, interrupt handler는 인터럽트 처리 도중 또 다른 인터럽트가 걸려 올 수 있으므로 최대한 빨리

처리가 될 수 있도록 코드 구성을 해 주어야 한다.

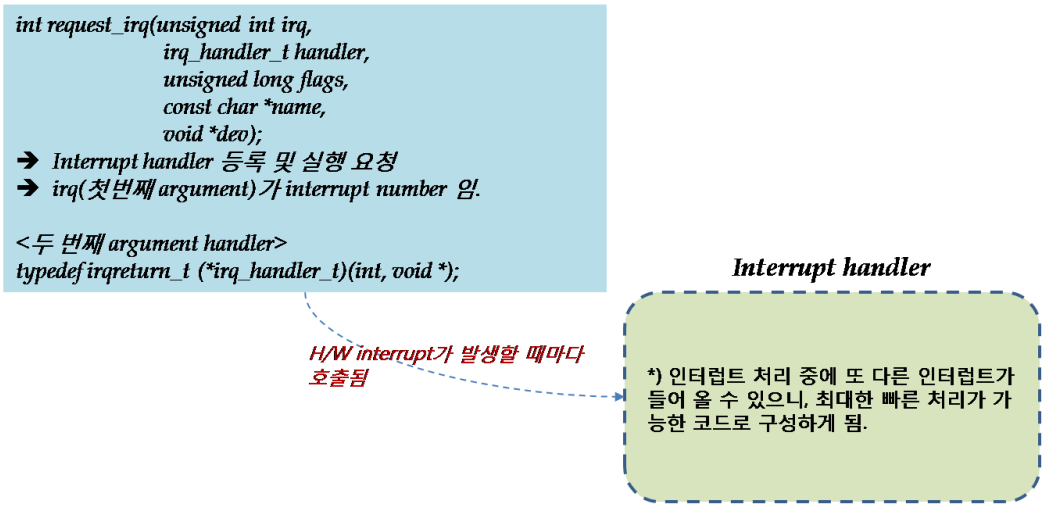

그림 2-20 인터럽트 핸들러 등록 및 실행

<일반적인 Interrupt handler 코드 구성>

1) 장치가 인터럽트를 계속 발생시킬 수 있도록 해준다.

2) 장치로부터 데이터를 읽거나, 장치에 데이터를 쓰는 작업을 수행한다.

3) 원하는 동작이 완료되기를 기다리고 있는 프로세스를 깨워준다.

- *대개 wake_up_interruptible(&module_queue) 함수 호출함.*

Kernel 프로그래머가 자주 사용하는 interrupt 관련 API를 소개해보면 다음과 같다.

[Top Half 관련 주요 API 소개 – include/linux/interrupt.h 참조]

static inline int __must_check ***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 *);*

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

void ***synchronize_irq****(unsigned int irq)*

- free_irq를 호출하기 전에 호출하는 함수로, 현재 처리 중인 interrupt handler가 동작을 완료하기를 기다려 줌.

void ***free_irq****(unsigned int irq, void *dev_id)*

- 인터럽트 handler 등록 해제 함수.

void ***disable_irq****(unsigned int irq)*

- 해당 IRQ 라인에 대한 interrupt 발생을 못하도록 함(completion을 기다림).

void ***disable_irq_nosync****(unsigned int irq)*

- Interrupt handler가 처리를 끝내도록 기다리지 않고 바로 return 함.

void ***enable_irq****(unsigned int irq)*

- 해당 IRQ 라인에 대한 interrupt 발생을 허용하도록 함.

[예제 코드 - drivers/mmc/host/omap_hsmmc.c]

```
static irqreturn_t omap_hsmmc_irq(int irq, void *dev_id)   //인터럽트 핸들러 함수     ③
{
    struct omap_hsmmc_host *host = dev_id;
    int status;

    status = OMAP_HSMMC_READ(host->base, STAT);
    while (status & INT_EN_MASK && host->req_in_progress) {
        omap_hsmmc_do_irq(host, status);

        /* Flush posted write */
        OMAP_HSMMC_WRITE(host->base, STAT, status);
        status = OMAP_HSMMC_READ(host->base, STAT);
    }

    return IRQ_HANDLED;
}
[...]
static int omap_hsmmc_probe(struct platform_device *pdev)     //probe() 함수     ①
{
    struct omap_mmc_platform_data *pdata = pdev->dev.platform_data;
    struct mmc_host *mmc;
    struct omap_hsmmc_host *host = NULL;
    int ret, irq;

    [...]
```

```
    /* Request IRQ for MMC operations */
    ret = request_irq(host->irq, omap_hsmmc_irq, 0, mmc_hostname(mmc), host);
      //인터럽트 핸들러 등록     ②
    if (ret) {
        dev_err(mmc_dev(host->mmc), "Unable to grab HSMMC IRQ₩n");
        goto err_irq;
    }
    [...]
}

static int omap_hsmmc_remove(struct platform_device *pdev)    //remove() 함수 – 드라이버 제거시 호출    ④
{
    struct omap_hsmmc_host *host = platform_get_drvdata(pdev);
    struct resource *res;

    [...]
    free_irq(host->irq, host);    //인터럽트 핸들러 등록 해제    ⑤
    if (mmc_slot(host).card_detect_irq)
        free_irq(mmc_slot(host).card_detect_irq, host);
    [...]
}
```

Shared Interrupt Handler

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 안 된다.

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 자

신의 device로 부터 interrupt가 발생했는지를 판단하여, 그렇지 않을 경우에는 재빨리 handler 루틴을 끝내야 한다.

Interrupt Enable/Disable 함수

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);
local_irq_restore(flags); // 저장된 상태 복구

enable_irq(irq);

- system의 모든 processor로 부터의 interrupt를 허용함
- [주의 사항] enable_irq/disable_irq는 항상 쌍으로 호출되어야 한다. 즉, disable_irq를 두 번 호출했으면 Enable_irq도 두 번 호출해 주어야 금지된 interrup가 해제된다.

local_irq_disable(); // 현재 processor 내부에서만 interrupt를 금지 시켜줌.
/* interrupts are disabled */
local_irq_enable();

이상의 내용을 표로 정리하면 다음과 같다.

표 2-1 Interrupt Enable/Disable 함수 정리

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.

3.2 SoftIRQ(Bottom Half)

Linux에는 두 종류의 software interrupt가 있다. 즉, bottom half 개념의 soft interrupt와 system call(SWI) 가 그것이다. 본 장에서 언급하는 내용은 bottom half 개념의 soft interrupt 이다. 그림 2-21에는 soft interrupt의 개념이 표현되어 있는데, soft interrupt handler는 hard interrupt handler가 처리하기에는 다소 지연 시간을 갖는 일을 별도로 처리하는 것으로 interrupt context에서 실행된다는 특징을 갖는다(반면에 앞으로 설명한 워크 큐는 process context에서 실행됨).

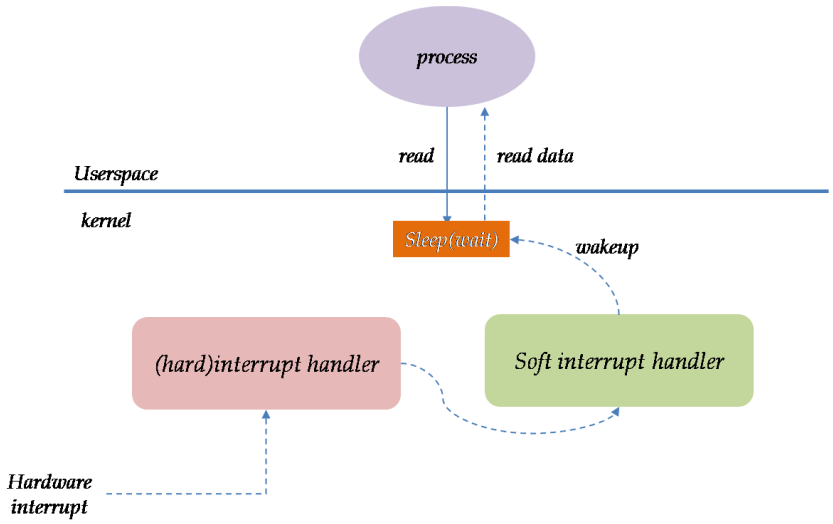

그림 2-21 SoftIRQ 개념(1) – hardware interrupt와의 차이

SoftIRQ를 구성하는 것으로는 High, Low Tasklet, Timer, Network(Tx/Rx), SCSI 지연 처리 등이며, 이 중 Tasklet에 관하여 집중적으로 살펴 보기로 한다.

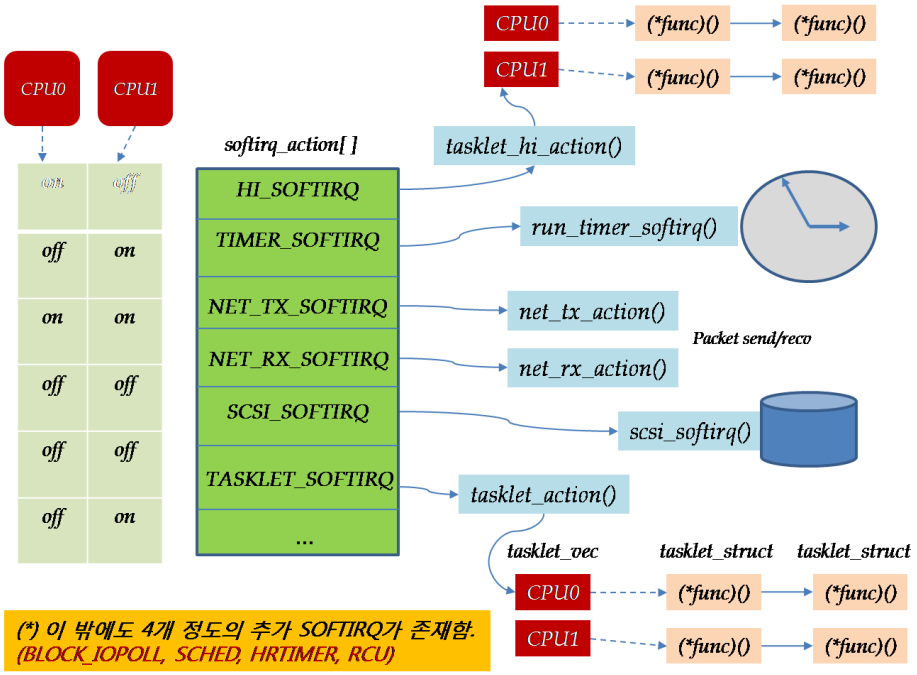

그림 2-22 SoftIRQ 개념(2)

3.3 태스크릿(tasklet)

Tasklet과 softirq의 동작 원리는 동일하다. 다만, softirq는 compile-time에 이미 내용(action)이 정해져 있으며, tasklet은 dynamic하게 등록할 수 있는 형태라 할 수 있다. Tasklet은 동시에 하나씩만 실행된다(count와 state 값을 활용). 이는 multi-processor 환경에서도 동일하게 적용된다. 그리고, tasklet은 task 개념과는 전혀 무관하며, 또한 work queue와는 달리 kernel thread를 필요로 하지 않는다(그만큼 간단한 작업을 처리한다고 보아야 할 듯 함).

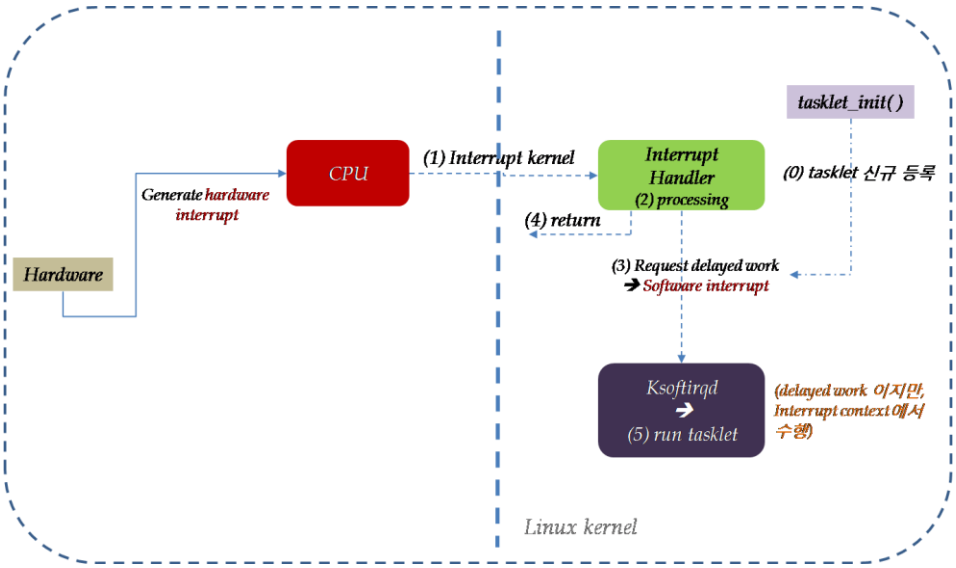

그림 2-23 태스크릿 개념도(1)

Tasklet 관련 코드를 구현하는 방식은 앞서 설명한 hard interrupt handler와 유사하다. Kernel programmer는 tasklet handler를 구현한 후, tasklet_init() 함수나 DECLARE_TASKLET 매크로를 이용해 등록하는 과정을 거친다. 이후, tasklet handler를 수행하고 싶은 시점이 되면, tasklet_schedule() 함수를 호출하여 주면 된다.

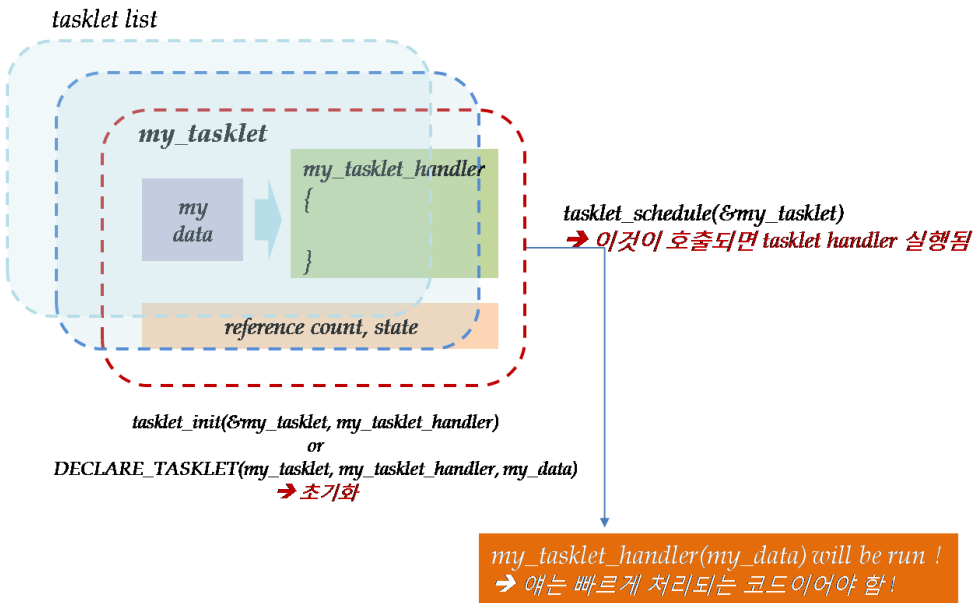

그림 2-24 태스크릿 개념도(2)

[Tasklet 관련 주요 API 소개 – include/linux/interrupt.h 참조]

void ***tasklet_init****(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data)*

혹은

DECLARE_TASKLET*(name, func, data)*

- *초기화*

static inline void ***tasklet_schedule****(struct tasklet_struct *t)*

- *이것이 호출되면 tasklet handler 실행됨*

static inline void ***tasklet_enable****(struct tasklet_struct *t)*

- *disable된 tasklet를 enable 시킬 때 사용*

static inline void ***tasklet_disable****(struct tasklet_struct *t)*

- *enable된 tasklet를 disable 시킬 때 사용*

void ***tasklet_kill****(struct tasklet_struct *t)*

- *tasklet를 pending queue에서 제거할 때 사용*

[예제 코드 - drivers/spi/spi-atmel.c]

```
struct atmel_spi {
    [...]
    struct tasklet_struct    tasklet;    //tasklet 변수 선언    ①
    [...]
};

/* Tasklet
 * Called from DMA callback + pio transfer and overrun IRQ.
 */
static void atmel_spi_tasklet_func(unsigned long data)    //tasklet handler 함수 수행    ⑦
{
    struct spi_master    *master = (struct spi_master *)data;
    struct atmel_spi     *as = spi_master_get_devdata(master);
    struct spi_message   *msg;
    struct spi_transfer *xfer;

    [...]
}

static irqreturn_t atmel_spi_pio_interrupt(int irq, void *dev_id)    //인터럽트 핸들러 함수    ⑤
{
    struct spi_master    *master = dev_id;
    struct atmel_spi     *as = spi_master_get_devdata(master);
    u32                  status, pending, imr;
    struct spi_transfer *xfer;
    int                  ret = IRQ_NONE;

    [...]
    if (pending & SPI_BIT(OVRES)) {
        [...]
        tasklet_schedule(&as->tasklet);
            //tasklet 핸들러 함수 실행. 이 코드에서는 인터럽트 발생 시마다 호출될 것임    ⑥
    } else if (pending & SPI_BIT(RDRF)) {
    [...]
```

```
}

static int atmel_spi_probe(struct platform_device *pdev)     //probe() 함수     ②
{
    struct resource        *regs;
    int                irq;
    struct clk          *clk;
    int                ret;
    struct spi_master      *master;
    struct atmel_spi       *as;
    [...]
    tasklet_init(&as->tasklet, atmel_spi_tasklet_func, (unsigned long)master);
      //tasklet 동적 초기화     ③

    ret = request_irq(irq, atmel_spi_pio_interrupt, 0, dev_name(&pdev->dev), master);
      //인터럽트 핸들러 등록     ④
    [...]
}

static int atmel_spi_remove(struct platform_device *pdev)     //remove() 함수     ⑧
{
    [...]
    tasklet_kill(&as->tasklet);     //tasklet를 pending queue에서 제거(해제)     ⑨
    [...]
}
```

아래 code는 softirq 및 tasklet을 실제로 처리해 주는 ksoftirqd kernel thread의 메인 루틴을 정리한 것이다. Softirq나 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 되었다. 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.

<ksoftirqd kernel thread>

```
for (;;) {
    if (!softirq_pending(cpu))        /* softirq/tasklet 요청이 없으면, sleep */
        schedule( );

    set_current_state(TASK_RUNNING);

    while (softirq_pending(cpu)) {
        do_softirq( );                /* softirq stack의 내용 실행 */
        if (need_resched( ))
                schedule( );
    }
    set_current_state(TASK_INTERRUPTIBLE);
}
```

<Tasklet scheduling 절차>

1) Tasklet의 상태가 TASKLET_STATE_SCHED인지 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(**Tasklet handler 수행 절차**)*

6) Interrupt를 이전 상태로 복구하고, return 한다.

<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 disable되어 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.

3.4 워크 큐(work queue)

워크 큐는 앞서 설명한 바와 같이 interrupt 발생 상황에서 지연 처리가 필요한 경우에 사용하거나, 특정 커널 혹은 디바이스 드라이버 내에서 주기적으로 반복하는 코드를 작성하고자 하는 경우에 사용하는 처리 방식이다.

<워크 큐의 주요 특징>

1) Work queue라고 하면, work, queue, worker thread의 세가지 요소를 통칭한다.
2) Work queue를 위해서는 반드시 worker thread가 필요하다.
3) 기 정의된 worker thread(events/0)를 사용하는 방식과 새로운 worker thread 및 queue를 만드는 방법(사용자 정의 워크 큐)이 존재한다.
4) Work queue에서 사용하는 work는 당연히 sleep이 가능하다.

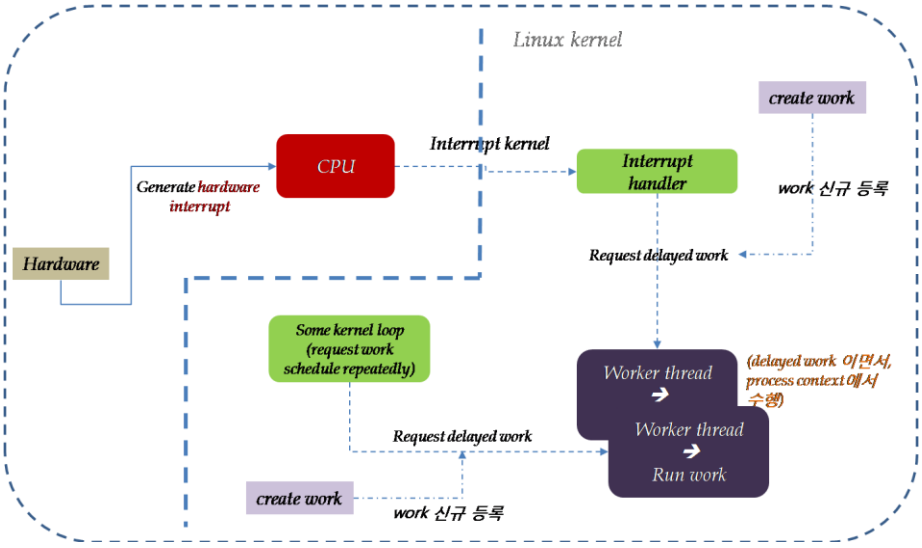

그림 2-25 워크 큐 개념도(1)

Old 워크 큐(cmwq) 소개

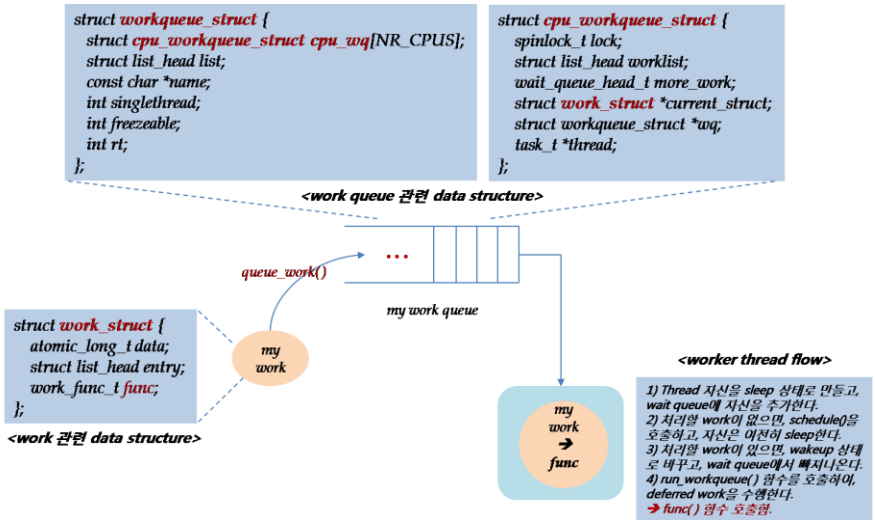

그림 2-26 워크 큐 개념도(2) - old data structure

워크 큐 관련 코드를 구현하는 방식은 앞서 설명한 hard interrupt handler나 Tasklet 등과 유사하다. Kernel programmer는 work function를 구현한 후, DECLARE_WORK 매크로나 INIT_WORK 매크로를 이용해 초기화(등록) 과정을 거친다. 이후, work function을 수행하고 싶은 시점이 되면, schedule_work() 함수나 schedule_delayed_work() 함수를 호출하여 주면 된다. 실제로 work function을 처리하는 코드는 worker thread로 CPU 스케쥴링이 되고 나면, 현재 등록되어 있는 work function을 차례로 실행해 주는 역할을 한다.

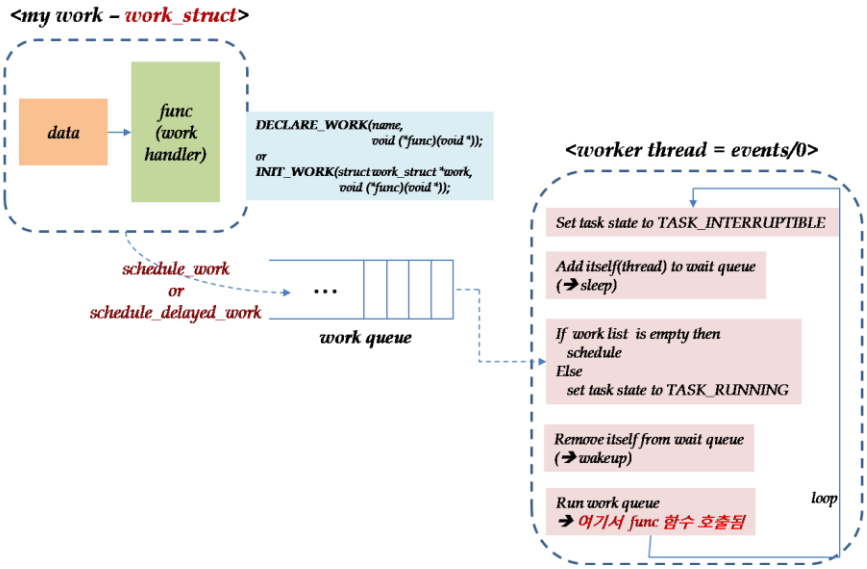

그림 2-27 워크 큐 개념도(3)

[기 정의된 Work Queue 관련 주요 API 소개 – include/linux/workqueue.h 참조]

static inline bool ***schedule_work****(struct work_struct *work)*

- *Global work queue에 work task를 추가함*

static inline bool ***schedule_delayed_work****(struct delayed_work *dwork, unsigned long delay)*

- *Delay 시간이 지나고 나서, work task를 global work queue에 추가시킴.*

void ***flush_scheduled_work****(void)*

- *work queue의 모든 엔트리를 처리(비움). 모든 entry가 실행됨. Entry를 취소하는 것이 아님에 주의해야 한다. 또한 schedule_delayed_work은 flush시키지 못함.*

bool ***cancel_delayed_work****(struct delayed_work *dwork)*

- *delayed work(엔트리)를 취소 시킴.*

```
struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
    /* target workqueue and CPU ->timer uses to queue ->work */
    struct workqueue_struct *wq;
    int cpu;
};
```

- *work과 timer를 묶어 새로운 data structure 정의 !*

사용자 정의 work queue를 생성하기 위해서는 create_workqueue() 함수를 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.

[사용자 정의 Work Queue 관련 주요 API 소개 – include/linux/workqueue.h 참조]

*#define **create_workqueue**(name) **alloc_workqueue**((name), WQ_MEM_RECLAIM, 1)*

- *사용자 정의 워크 큐 및 woker thread를 생성시켜 줌.*

*static inline bool **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.*

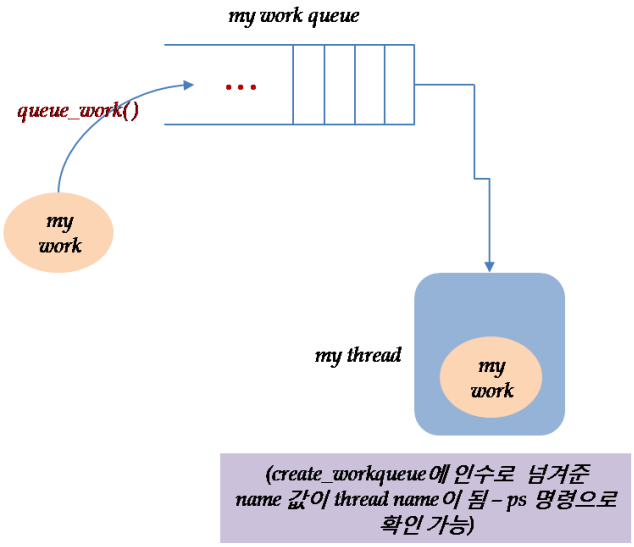

그림 2-28 사용자 정의 워크 큐

[예제 코드 - drivers/mmc/core/core.c]

```
static struct workqueue_struct *workqueue;    //workqueue 변수 선언    ①

static int mmc_schedule_delayed_work(struct delayed_work *work, unsigned long delay)
{
    return queue_delayed_work(workqueue, work, delay);
      // 일정 시간 경과 후, 사용자 정의 work을 사용자 정의 work queue에 넣고,
```

```
        // schedule 요청함      ③
}

static void mmc_flush_scheduled_work(void)
{
      flush_workqueue(workqueue);
        // 사용자 정의 work queue에 있는 모든 work을 처리하여, queue를 비우도록 요청      ④
}

void mmc_detect_change(struct mmc_host *host, unsigned long delay)
{
        [...]
        mmc_schedule_delayed_work(&host->detect, delay);
}

void mmc_rescan(struct work_struct *work)
{
        [...]
out:
        if (host->caps & MMC_CAP_NEEDS_POLL)
                mmc_schedule_delayed_work(&host->detect, HZ);
}

void mmc_stop_host(struct mmc_host *host)
{
        [...]
        mmc_flush_scheduled_work();
        [...]
}

int mmc_suspend_host(struct mmc_host *host)
{
        [...]
        mmc_flush_scheduled_work();
}

static int __init mmc_init(void)
{
```

```
        int ret;
        workqueue = alloc_ordered_workqueue("kmmcd", 0);
            //사용자 정의 workqueue 생성     ②
        if (!workqueue)
                return -ENOMEM;
        [...]
}

static void __exit mmc_exit(void)
{
        [...]
        destroy_workqueue(workqueue);     //workqueue 제거     ⑤
}
```

신규 워크 큐(cmwq) 소개

새로운 워크 큐(cmwq)는 workqueue 당 worker thread를 할당하는 것이 아니라, cpu 별로 worker thread를 두도록 하였으며, 현재 work이 sleep으로 들어 갈 경우, 그냥 대기하는 것이 아니라, 다른 worker thread를 생성하여 이를 처리하도록 함으로써 concurrency를 높였다. 만일 새로운 work이 없고, idle한 thread가 5분간 지속되면, 해당 thread를 제거함으로써 resource를 효율적으로 관리하게 된다.

<cmwq의 설계 목표>

1) 기존 work queue API와의 호환성 유지

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

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

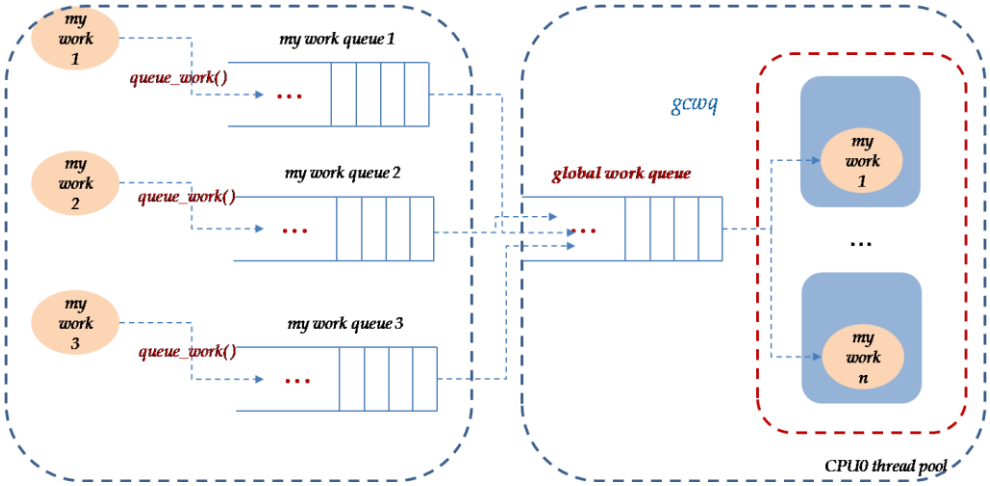

그림 2-29 신규 워크 큐(1)

<워크 큐 동작 방식>

<Kernel Programmer>

1) 처리할 work 정의

2) 신규 work queue 생성 요청 혹은 default work queue 사용

3) Work 처리 요청(schedule)

<Kernel Codes>

4) 사용할 gcwq를 결정함(찾아냄)

- *이미 해당 work이 처리 중이었다면, 이를 처리하던 worker를 찾아내고,*
- *그렇지 않다면, gcwq를 할당함.*

5) gcwq가 결정되었으면, 이로부터 cwq & queue를 다시 구함.

6) cwq 및 work 정보를 토대로 work을 queue에 추가함.

- *data 부분을 실제 연결하고,*
- *list에 entry를 추가함.*
- *worker pool에서 idle한 첫 번째 worker를 찾아낸 후,*
- *worker를 깨어 work(task)를 실행 시킴.*

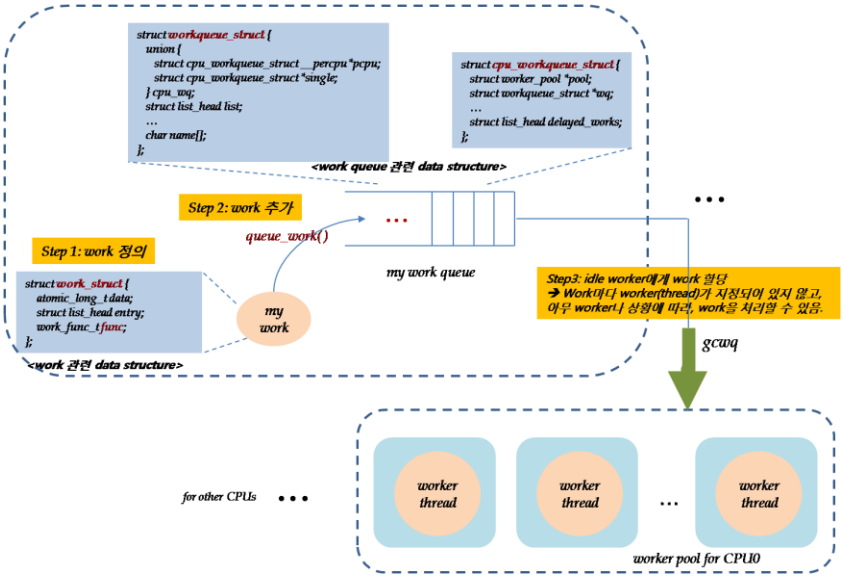

그림 2-30 신규 워크 큐(2)

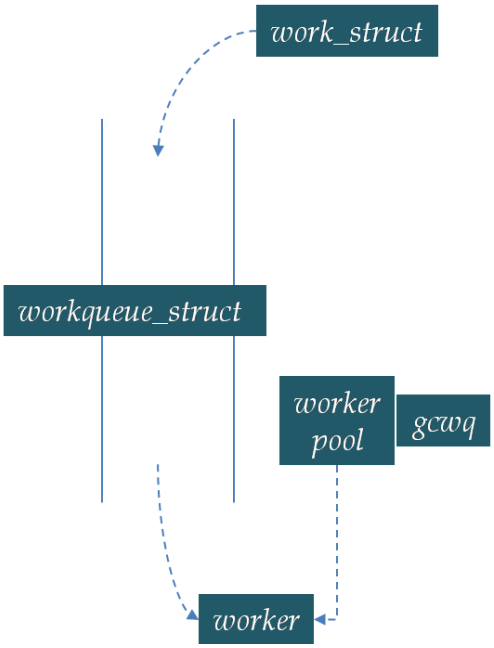

그림 2-31 신규 워크 큐(3)

Kernel programmer가 작성한 work과 kernel이 생성하는 worker thread는 처리 여부 및 동작 여부에 따라 아래 그림과 같이 분류가 가능하다.

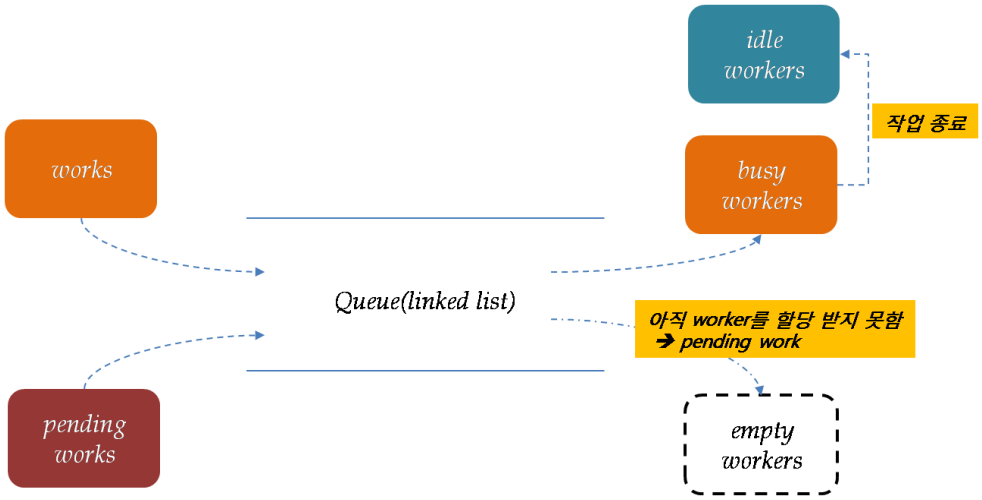

그림 2-32 신규 워크 큐(4)

아래 그림은 gcwq(global per-cpu workqueue)에서 생성하는 worker thread를 화면 capture한 것이다. kworker/0:0의 첫 번째 숫자는 cpu를 나타내고, 두 번째 숫자는 thread id를 의미한다. 앞서도 설명한 바와 같이, 기존 방식의 경우 workqueue 별로 thread를 생성하므로, thread가 무한정(?)늘어나는 형태가 되지만, 새로 도입된 cmwq에서는 위의 그림에서와 같이 kworker thread가 제한된 범위 내에서 늘었다 줄었다를 반복하게 되므로, 보다 효율적이라고 할 수 있다.

```
   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]
```

그림 2-33 worker thread

지금까지 설명한 내용을 토대로 일반적인 work queue를 사용하는 절차를 끝으로 워크 큐에 대한 설명을 마무리 하도록 하겠다.

[일반적인 Work Queue 사용 절차]

1) struct delayed_work my_work;

- *delayed work 변수 선언*

2) if (delayed_work_pending(&my_work))

cancel_delayed_work_sync(&my_work);

- *대기중인 delayed work을 취소하는 경우 사용*
- *새로운 요청을 하기 전에, 보통 기존에 요청한 내용을 취소함.*

3) 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.*
- ***<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.*

4) cancel_delayed_work_sync(my_work);

- *대기중인 delayed work을 취소 요청, driver 종료 시 호출*
- *드라이버 동작을 종료할 경우, 기존에 요청한 work을 취소하도록 함.*

```
5) static void my_work_func(struct work_struct *work)
{
        /* ... */
}
```

- *work function 정의*
- *이 함수가 worker thread에서 수행됨.*

6) INIT_DELAYED_WORK(&my_work, my_work_func);

- *work 함수 초기화, driver 시작(init 혹은 probe 단계) 시, 호출*
- *delayed work의 work function 초기화*

3.5 커널 쓰레드(Kernel Thread)

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_struct 내의 mm pointer가 NULL임). 그러나, kernel thread는 user process와 마찬가지로 schedule 가능하며, 다른 kernel thread 혹은 interrupt handler 등에 의해 선점(preemptible)될 수 있다. 단, user-process에 의해 선점되지는 않는다. 사용자 정의 kernel thread는 kthreadd(parent of kernel threads)에 의해 추가 생성(fork)된다.

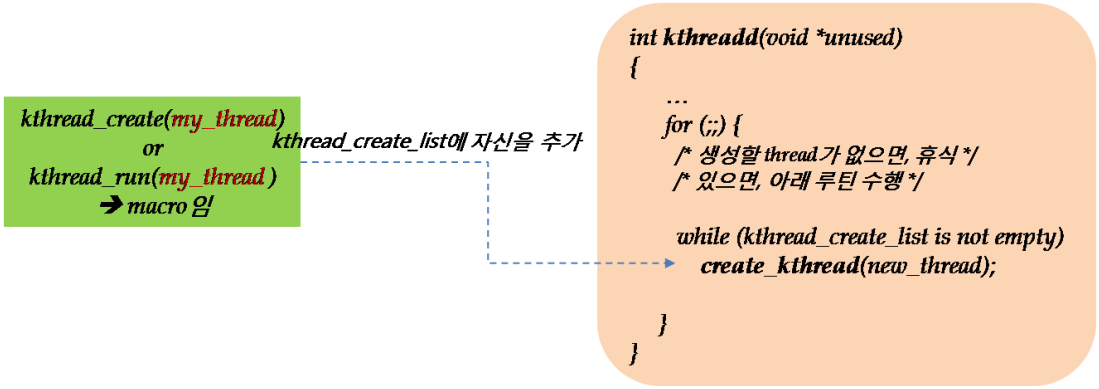

그림 2-34 kernel thread의 개념

create_kthread() 함수 내에서 사용자가 등록한 my_thread() 함수를 수행하는 과정을 코드로 표시하면 다음과 같다.

```
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);          /* 사용자가 등록한 thread function (my_thread) 수행 ! */
        [...]
        do_exit(ret);
}
```

Kernel thread를 생성하는 방법 및 실제 이 kernel thread가 실행되는 방식을 그림 2-35에 표현하였다.

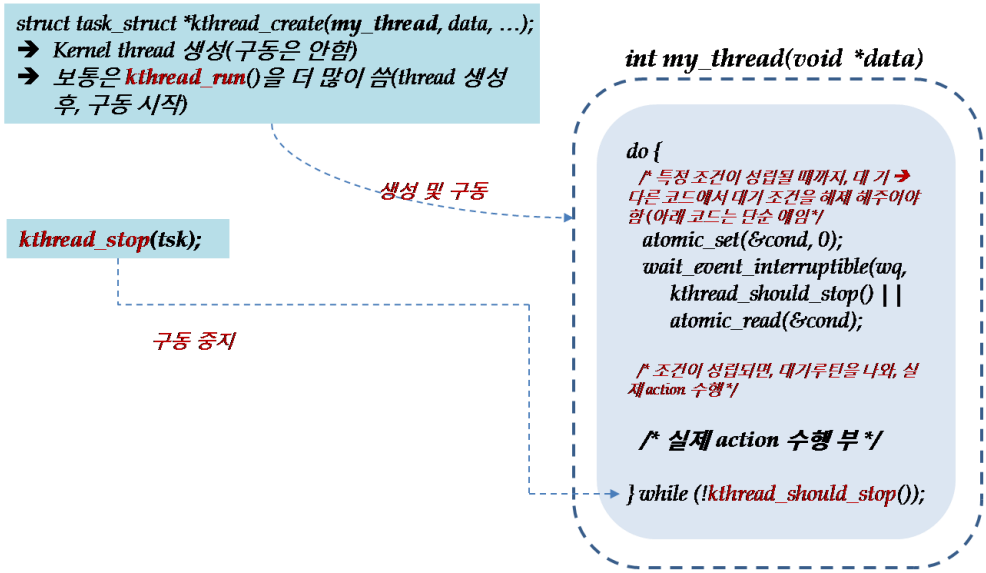

그림 2-35 kernel thread 생성 방법

[Kernel Thread 관련 주요 API 소개 – include/linux/kthread.h 참조]

kthread_run*(threadfn, data, namefmt, ...)*

- *kernel thread를 만들고, thread를 깨워줌*

*#define **kthread_create**(threadfn, data, namefmt, arg...) kthread_create_on_node(threadfn, data, -1, namefmt, ##arg)*

- *kernel thread를 만듦(sleeping 상태로 있음)*

*void **kthread_bind**(struct task_struct *p, unsigned int cpu)*

- *kernel thread를 특정 CPU에 bind 시켜 줌.*

*int **kthread_stop**(struct task_struct *k)*

- *kthread_create()로 만든 thread를 중지할 때 사용함. kthread_should_stop을 위한 조건을 설정해 줌.*

*bool **kthread_should_stop**(void)*

- *kernel thread 루틴을 멈추기 위한 조건 검사 함수.*

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.

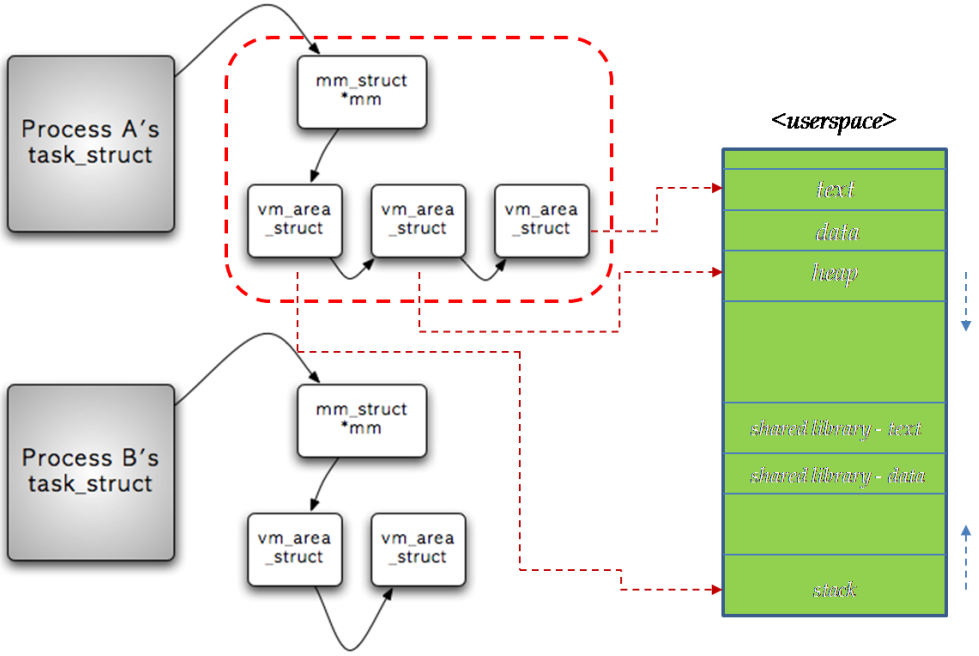

그림 2-36 kernel thread의 task_struct [다시 그려야 함]

[예제 코드 - drivers/net/wireless/brcm80211/brcmfmac/dhd_sdio.c]

```
struct brcmf_sdio {
    [...]
    struct task_struct *watchdog_tsk;    //thread를 위한 task_struct 변수 선언    ①
};

static void brcmf_sdbrcm_bus_stop(struct device *dev)    //stop 함수    ⑥
{
    [...]
    if (bus->watchdog_tsk) {
        send_sig(SIGTERM, bus->watchdog_tsk, 1);
        kthread_stop(bus->watchdog_tsk);    //thread를 중지시킴    ⑦
        bus->watchdog_tsk = NULL;
    }
    [...]
}

static int brcmf_sdbrcm_watchdog_thread(void *data)    //thread 함수    ④
{
```

```
    struct brcmf_sdio *bus = (struct brcmf_sdio *)data;

    allow_signal(SIGTERM);
    /* Run until signal received */
    while (1) {
        if (kthread_should_stop())    // kernel thread 루틴을 멈추기 위한 조건 검사    ⑤
            break;
        if (!wait_for_completion_interruptible(&bus->watchdog_wait)) {
            brcmf_sdbrcm_bus_watchdog(bus);
            /* Count the tick for reference */
            bus->sdcnt.tickcnt++;
        } else
            break;
    }
    return 0;
}

void *brcmf_sdbrcm_probe(u32 regsva, struct brcmf_sdio_dev *sdiodev)    //probe() 함수    ②
{
    int ret;
    struct brcmf_sdio *bus;
    [...]
    bus->watchdog_tsk =
                kthread_run(brcmf_sdbrcm_watchdog_thread, bus, "brcmf_watchdog");
      // kernel thread를 만들고, thread를 깨워줌    ③
    [...]
}
```

3.6 인터럽트 쓰레드(threaded IRQ)

이제부터는 Interrupt Thread의 개념과 필요성을 소개하고자 한다.

<Interrupt Thread>

1) Interrupt는 kernel 입장에서는 최대 latency의 근원이다.

- *이를 줄이기 위해 kernel thread에게 일을 넘겨주고, 즉시 return 함.*
- *이건 기존 bottom half와 차이가 없음. 실제 차이점은, 높은 thread에게 우선 순위를 부*

여하여, preemptible thread(process) context에서 동작하도록 하는 것임.

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.*

인터럽트 thread는 2.6.30 kernel 부터 소개된 기법(Real-time kernel tree에서 합류함)으로, response time을 줄이기 위해, 우선 순위가 높은 interrupt 요청 시 context switching이 일어난다. Interrupt 발생 시, hardware interrupt 방식으로 처리할지 Thread 방식으로 처리할지 결정하게 되는데, IRQ_WAKE_THREAD를 return하면, thread 방식으로 처리(IRQ thread를 깨우고, thread_fn을 실행함)하고, 그렇지 않으면 기존 hard interrupt handler로 동작하게 된다. 또한 handler가 NULL이고 thread_fn이 NULL이 아니면, 무조건 Threaded interrupt 방식으로 처리하게 된다. 이 방식은 앞서 소개한 tasklet 및 work queue의 존재를 위협할 수 있는 방식으로 인식되고 있다.

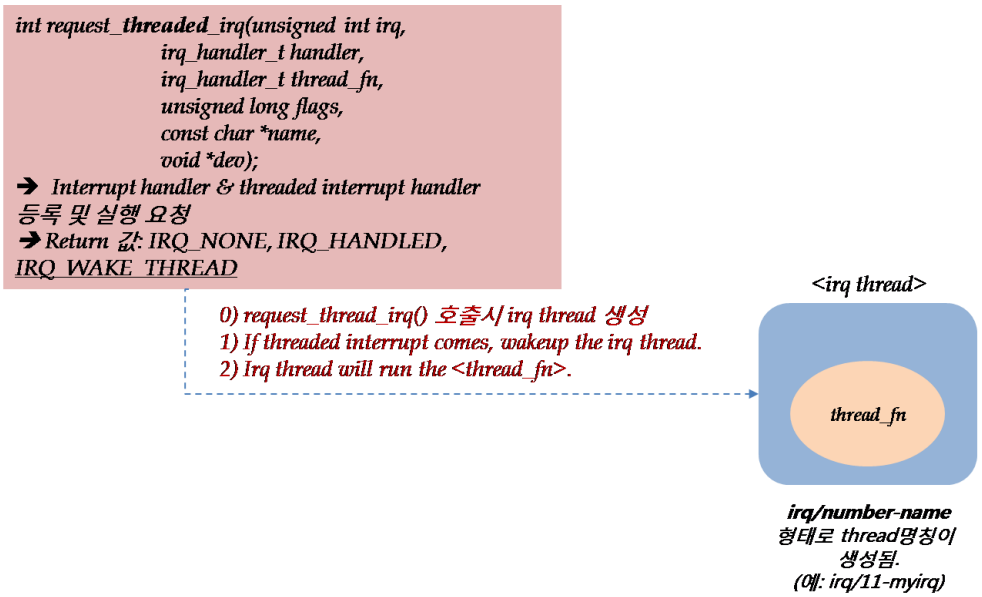

그림 2-37 인터럽트 thread 핸들러 등록 및 실행

IRQ thread가 실제로 어떻게 동작하는지를 코드를 통해 살펴보고자 하겠다. 아래 코드는 앞서 4장에서 이미 언급한 바 있는 handle_irq_event_percpu() 함수와도 밀접히 연관되어 있는데, 아래 코드를 제대로 이해하기 위해서는 사전에 몇가지를 숙지하고 있어야 할 필요가 있다.

0) 사용자는 request_threaded_irq() 함수를 써서 인터럽트 쓰레드 핸들러 함수를 등록한다.
1) 실제 hardware interrupt가 발생하여, irq/handle.c의 handle_irq_event_percpu() 함수가 동작하게 된다.
2) 1단계에서 인터럽트 쓰레드 핸들러를 등록한 상황이므로, handle_irq_event_percpu() 함수 내의 아래 문장이 실행된다.

```
case IRQ_WAKE_THREAD:
        irq_wake_thread(desc, action);
```

3) irq_wake_thread() 함수는 IRQTF_RUNTHREAD flasg를 켜주고, IRQ thread를 깨워준다.
*4-6) 마침내, irq/manage.c의 irq_thread() 함수가 동작(**TASK_RUNNING**)하여, 1에서 등록한 인터럽트 쓰레드 함수(**thread_fn**)를 호출되게 된다.*

IRQ handler 처리 과정 - kernel/irq/handle.c에서 발췌

```
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)    ①
{
    ...
    do {
        irqreturn_t res;

        res = action->handler(...);            /* request_irq()로 등록한 irq handler 실행 – 4장 */
        WARN_ONCE(!irqs_disabled(), ....);
        /* local irq가 disable되어 있지 않으면, warning message 출력 */
        /* interrupt 처리 중에는 다른 interrupt가 들어오면 안됨. */

        switch (res) {
        case IRQ_WAKE_THREAD:          // interrupt thread 이면    ②
            irq_wake_thread(irq, action)
              // interrupt thread를 깨워줌.
              // 이 함수에서 IRQTF_RUNTHREAD flag를 켜주고, thread를 깨워줌    ③

        case IRQ_HANDLED:
            flags |= action->flags;

        default:
            break;
        }
        action = action->next;          /* 다음 action 선택 */
                                        /* 같은 interrupt line, 복수개의 handler 등록 시 사용 */
```

```
    } while (action);
    ...
}
```

IRQ thread 동작 과정 분석 - kernel/irq/manage.c에서 발췌

```
static irqreturn_t irq_thread_fn(struct irq_desc *desc, struct irqaction *action)
{
    irqreturn_t ret;
    ret = action->thread_fn(action->irq, action->dev_id);
      //request_threaded_irq()로 등록한 인터럽트 핸들러 함수 호출    ⑥
    irq_finalize_oneshot(desc, action);
    return ret;
}

static int irq_wait_for_interrupt(struct irqaction *action)
{
    set_current_state(TASK_INTERRUPTIBLE);
    while (!kthread_should_stop()) {
        if (test_and_clear_bit(IRQTF_RUNTHREAD, &action->thread_flags)) {
            // IRQTF_RUNTHREAD  - interrupt handler thread가 구동해야 함을 알려 줌.
            // 이 flag는 irq_wake_thread() 함수에서 켜주며, irq_wake_thread() 함수는
            // handle_irq_event_percpu() 함수에서 호출해 줌 - 4장 참조
                __set_current_state(TASK_RUNNING);
                return 0;
        }
        schedule();   // IRQTF_RUNTHREAD가 켜져 있지 않으며, 다른 task를 schedule 함
        set_current_state(TASK_INTERRUPTIBLE);
    }
    __set_current_state(TASK_RUNNING);
    return -1;
}

static int irq_thread(void *data)
{
    struct callback_head on_exit_work;
    static const struct sched_param param = {
        .sched_priority = MAX_USER_RT_PRIO/2,
```

```
    };
    struct irqaction *action = data;
    struct irq_desc *desc = irq_to_desc(action->irq);
    irqreturn_t (*handler_fn)(struct irq_desc *desc, struct irqaction *action);

    if (force_irqthreads && test_bit(IRQTF_FORCED_THREAD, &action->thread_flags))
        handler_fn = irq_forced_thread_fn;
    else
        handler_fn = irq_thread_fn;

    sched_setscheduler(current, SCHED_FIFO, &param);
      //current task(= irq thread)의 우선 순위 지정(우선순위 값 50으로 지정)

    init_task_work(&on_exit_work, irq_thread_dtor);
    task_work_add(current, &on_exit_work, false);
    irq_thread_check_affinity(desc, action);

    while (!irq_wait_for_interrupt(action)) {   //interrupt thread를 구동해야 할지 조건 검사 ④
        irqreturn_t action_ret;
        irq_thread_check_affinity(desc, action);

        action_ret = handler_fn(desc, action);
            //irq thread function(= interrupt handler 함수) 호출     ⑤

        if (!noirqdebug)
            note_interrupt(action->irq, desc, action_ret);
        wake_threads_waitq(desc);
    }

    task_work_cancel(current, irq_thread_dtor);
    return 0;
}
```

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

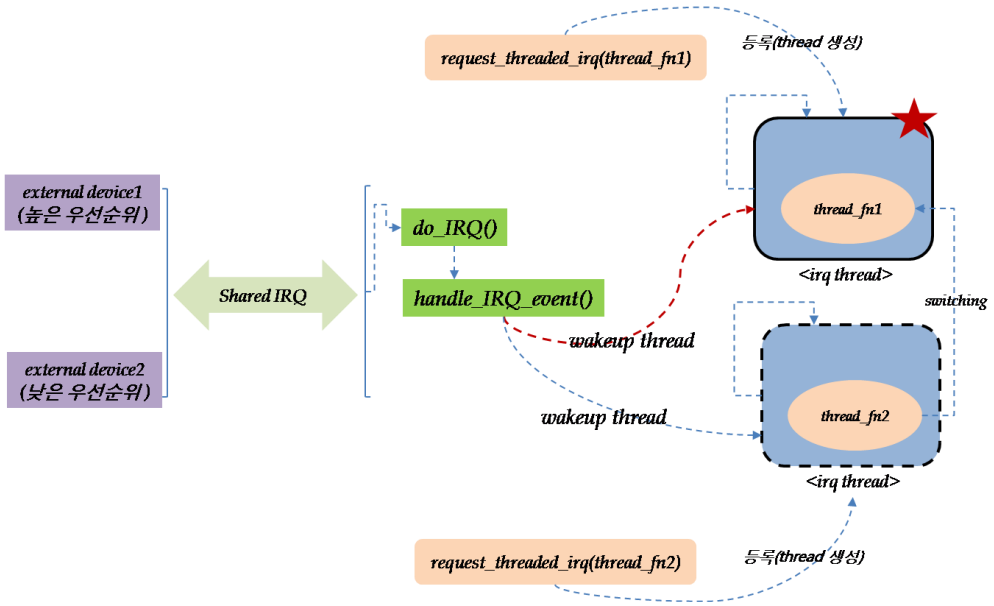

그림 2-38 인터럽트 thread 핸들러의 우선 순위 선점 [다시 그려야 함]

[예제 코드 - drivers/input/touchscreen/wm831x-ts.c]

```
static irqreturn_t wm831x_ts_data_irq(int irq, void *irq_data)
    //인터럽트 핸들러 쓰레드 수행 – 수행 절차는 위의 코드 분석 내용 참조    ③
{
    struct wm831x_ts *wm831x_ts = irq_data;
    struct wm831x *wm831x = wm831x_ts->wm831x;
    static int data_types[] = { ABS_X, ABS_Y, ABS_PRESSURE };
    u16 data[3];
    int count;
    int i, ret;

    [...]
    input_sync(wm831x_ts->input_dev);

    return IRQ_HANDLED;
}

static int wm831x_ts_probe(struct platform_device *pdev)    //probe() 함수    ①
{
    [...]
    error = request_threaded_irq(wm831x_ts->data_irq, NULL, wm831x_ts_data_irq,
                                 irqf | IRQF_ONESHOT, "Touchscreen data", wm831x_ts);
```

```
                //인터럽트 쓰레드 등록     ②
    if (error) {
        dev_err(&pdev->dev, "Failed to request data IRQ %d: %d₩n", wm831x_ts->data_irq, error);
        goto err_alloc;
    }
    disable_irq(wm831x_ts->data_irq);
    [...]
}

static int wm831x_ts_remove(struct platform_device *pdev)     //remove() 함수     ④
{
    [...]
    free_irq(wm831x_ts->data_irq, wm831x_ts);     //인터럽트 핸들러 해제(제거)     ⑤
    return 0;
}
```

4. 타이머

앞서 설명한 bottom half의 목적은 work을 단순히 delay시키는데 있는 것이 아니라, 지금 당장 work을 실행하지 않는데 있다. 한편 timer는 일정한 시간만큼 work을 delay시키는데 목적이 있다. Timer는 timer interrupt를 통해 동작하는 방식을 취한다(software interrupt). 즉, 처리하려는 function을 준비한 후, 정해진 시간이 지나면 timer interrupt가 발생하여 해당 function을 처리하는 구조이다. timer는 cyclic(무한 반복) 구조가 아니므로, time이 경과하면 timer function이 실행되고, 해당 timer는 제거되는 특징이 있다.

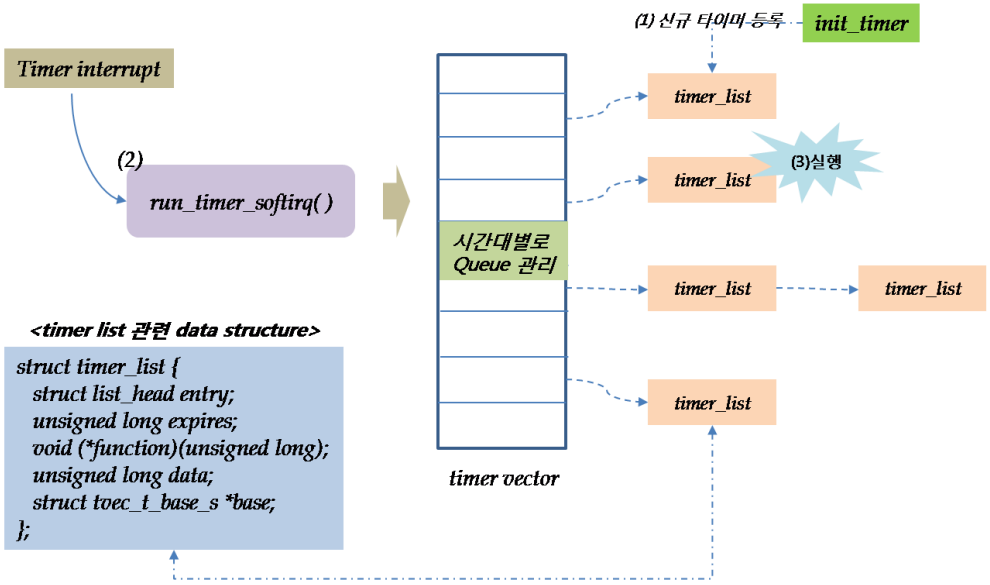

그림 2-39 타이머의 동작 원리

Timer를 구현하는 방식은 간단하다.

1) Kernel programmer는 timer function(handler)을 정의한다.

2) init_timer() 함수나 TIMER_INITIALIZER 매크로를 이용하여 struct timer_list my_timer를 초기화한다.

3) 1)에서 정의한 timer_function을 my_timer에 대입한다.

4) add_timer() 함수를 사용하여 timer를 구동시킨다.

5) 코드 수행 중에 timer expiration 값을 바꿔 주고 싶을 때는 mod_timer() 함수를 호출해 준다.

이를 그림으로 정리해 보면 다음과 같다.

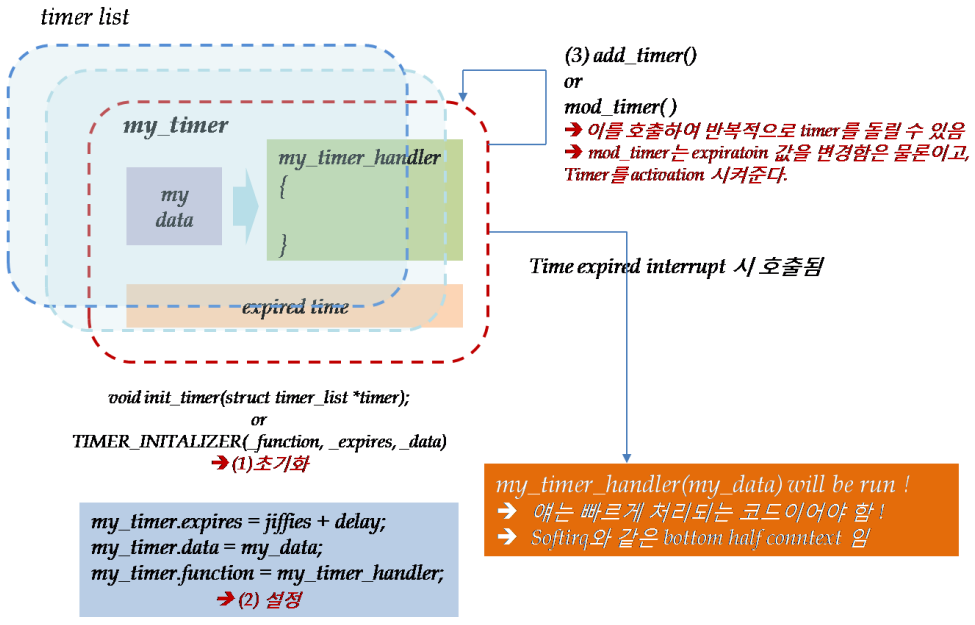

그림 2-40 타이머 구현 및 실행 과정

[Timer 관련 주요 API 소개 – include/linux/timer.h 참조]

*#define **init_timer**(timer) __init_timer((timer), 0)*

- *매크로 임.*
- *동적 초기화*

***TIMER_INITIALIZER**(_function, _expires, _data)*

- *매크로 임.*
- *정적 초기화*

*void **add_timer**(struct timer_list *timer);*

- *timer를 시작함. Timer를 활성화 시켜 줌.*

*int **mod_timer**(struct timer_list *timer, unsigned long expires);*

- *timer의 timeout 값을 갱신하고, timer를 활성화 시켜 줌.*

*int **del_timer**(struct timer_list * timer);*

- *timer를 비활성화 시킨다.*

*int **del_timer_sync**(struct timer_list *timer);*

- *timer를 비활성화 시킨다. 단, timer handler 함수가 끝날때까지 기다려 줌.*

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

[예제 코드 - drivers/nfc/pn533.c]

```
struct pn533 {
    [...]
    struct timer_list listen_timer;    //timer 변수 선언, See include/linux/timer.h 파일 참조    ①
    [...]
};

static void pn533_listen_mode_timer(unsigned long data)    //timer 핸들러 함수 수행    ⑥
{
    struct pn533 *dev = (struct pn533 *)data;
    nfc_dev_dbg(&dev->interface->dev, "Listen mode timeout");
```

```
    dev->cancel_listen = 1;
    pn533_poll_next_mod(dev);
    queue_work(dev->wq, &dev->poll_work);
}

static int pn533_poll_complete(struct pn533 *dev, void *arg, struct sk_buff *resp)
{
    [...]
    if (cur_mod->len == 0) { /* Target mode */
        del_timer(&dev->listen_timer);     //timer 제거
        rc = pn533_init_target_complete(dev, resp);
        goto done;
    }
    [...]
}

static void pn533_wq_poll(struct work_struct *work)
{
    [...]
    if (cur_mod->len == 0 && dev->poll_mod_count > 1)
        mod_timer(&dev->listen_timer, jiffies + PN533_LISTEN_TIME * HZ);
            //timer 갱신, timer를 활성화 시켜 줌.    ⑤
    return;
}

static void pn533_stop_poll(struct nfc_dev *nfc_dev)
{
    [...]
    del_timer(&dev->listen_timer);       //timer 제거
    [...]
}

static int pn533_probe(struct usb_interface *interface, const struct usb_device_id *id)
     //probe함수     ②
{
    [...]
    init_timer(&dev->listen_timer);     //timer 동적 초기화     ③
    dev->listen_timer.data = (unsigned long) dev;
```

```
    dev->listen_timer.function = pn533_listen_mode_timer;    //timer 핸들러 초기화    ④
    [...]
}

static void pn533_disconnect(struct usb_interface *interface)
{
    [...]
    del_timer(&dev->listen_timer);      //timer 제거
    [...]
}
```

Timer 관련하여 몇 가지 용어를 소개하고자 한다.

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*

<schedule_timeout (timeout) 함수 소개 - kernel/timer.c에서 발췌>

현재 실행 중인 task에 대해 delay를 줄 수 있는 보다 효과적인 방법으로 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 해준다. schedule_timeout()의 내부는 timer와 schedule 함수로 구성되어 있다.

```
signed long __sched schedule_timeout(signed long timeout)
{
    struct timer_list timer;
```

```
    unsigned long expire;

    switch (timeout)
    {
        case MAX_SCHEDULE_TIMEOUT:
            schedule();
            goto out;
        default:
            [...]
    }

    expire = timeout + jiffies;

    setup_timer_on_stack(&timer, process_timeout, (unsigned long)current);
    __mod_timer(&timer, expire, false, TIMER_NOT_PINNED);
    schedule();
    del_singleshot_timer_sync(&timer);

    destroy_timer_on_stack(&timer);
    timeout = expire - jiffies;
out:
    return timeout < 0 ? 0 : timeout;
}
```

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

process_timeout( )
sleep_on_timeout( )
interruptible_sleep_on_timeout( )

msleep 함수 내부를 잠시 살펴 보면, 내부적으로는 schedule_timeout 함수를 사용하고 있다. 다만 차이점이 있다면, uninterruptible로 되어 있어, 주어진 시간만큼은 확실이 sleep 상태에 있게 된다. 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 있게 된다.

<msleep( ) 함수 – kernel/timer.c>

```
void msleep(unsigned int msecs)
```

```
{
        unsigned long timeout = msecs_to_jiffies(msecs) + 1;
        while (timeout)
                timeout = schedule_timeout_uninterruptible(timeout);
}
```

Hrtimer(high resolution timer)는 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.

[예제 코드 - hrtimer]

```
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;
}
```

<TODO – HRTIMER API 정리해야 함>

본 서의 내용 중 일부는 Free Electrons의 문서[참고문헌 8-10]를 참고하였음을 밝힌다. 특히 그림 중 일부는 그대로 복사하여 사용하였다.

References

[1] Linux Kernel Development(3rd edition), Robert Love, Addison Wesley

[2] Writing Linux Device Drivers, Jerry Cooperstein

[3] Essential Linux Device Drivers, Sreekrishnan Venkateswaran, Prentice Hall

[4] Linux kernel 2.6 구조와 원리, 이영희 역, 한빛미디어

[5] 코드로 알아보는 ARM 리눅스 커널, 노서영 외 5인, Jpub

[6] 리눅스 커널 내부 구조, 백승재, 최종무, 교학사

[7] Professional Linux Kernel Architecture, Wolfgang Mauerer, Wrox

[8] Understanding Linux Kernel(3rd endition), Daniel P. Bovet, Marco Cesati, O'Reilly

[9] Android Kernel Hacks1/2, Chunghan Yi, www.kandroid.org

[10] Linux Kernel and Driver Development Training, Gregory Clement, Michael Opdenacker, Maxime Ripard, S_ebastien Jan, Thomas Petazzoni, Free Electrons

[11] Embedded Linux system development, Gregory Clement, Michael Opdenacker, Maxime Ripard, Thomas Petazzoni, Free Electrons

[12] Linux Kernel architecture for device drivers, Thomas Petazzoni Free Electronics

[13] The sysfs Filesystem, Patrick Mochel, mochel@digitalimplant.org

[14] InterruptThreads-Slides_Anderson.pdf, Mike Anderson, mike@theptrgroup.com