Linux Kernel Training. Lecture 7

Kthreads Overview
Interrupt Handling

Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

April 10, 2020. GlobalLogic

Kthreads Overview

Kthreads API

See `linux/kthread.h`, `linux/sched.h`

```
struct task_struct *kthread_create(int (*threadfn)(void *data), void *data,
                                   const char namefmt[], ...);
```

`threadfn`	the function to run until signal_pending(current).
`data`	data ptr for @threadfn.
`namefmt, ...`	printf-style name for the thread (format and arguments)

- Task created in a sleeping state.
- Has to be woken up by
    - int **wake_up_process**(struct task_struct *p)

```
#define kthread_run(threadfn, data, namefmt, ...)
```

Creates and runs thread, the same parameters as above

Kthreads API

```
/* bind thread to selected CPU
 * @p: thread created by kthread_create().
 * @cpu: cpu (might not be online, must be possible) for @k to run on.
 */
void kthread_bind(struct task_struct *p, unsigned int cpu)

/* Do not kill thread -- ask it to stop
 * @k: thread created by kthread_create().
 */
int kthread_stop(struct task_struct *k)

/* In thread -- check if stop is requested */
int kthread_should_stop(void);
```

Kthreads templates

```
/* thread for single job */
int thread_func(void *data)
{
    struct thread_params *params = data;

    initialization();

    do_your_job();

    signal_completion();

    return status;
}
```

```
/* thread for repetitive job */
int thread_func(void *data)
{
    struct thread_params *params = data;

    initialization();

    while (!kthread_should_stop()) {
        wait_input_ready();
        process_data();
        write_to_output();
    }

    signal_completion();

    return status;
}
```

Kthread worker

```
struct kthread_worker {
        unsigned int            flags;
        spinlock_t              lock;
        struct list_head        work_list;
        struct list_head        delayed_work_list;
        struct task_struct      *task;
        struct kthread_work     *current_work;
};

struct kthread_work {
        struct list_head        node;
        kthread_work_func_t     func;
        struct kthread_worker *worker;
        int                     canceling;
};

struct kthread_delayed_work {
        struct kthread_work work;
        struct timer_list timer;
};
```

Kthread worker

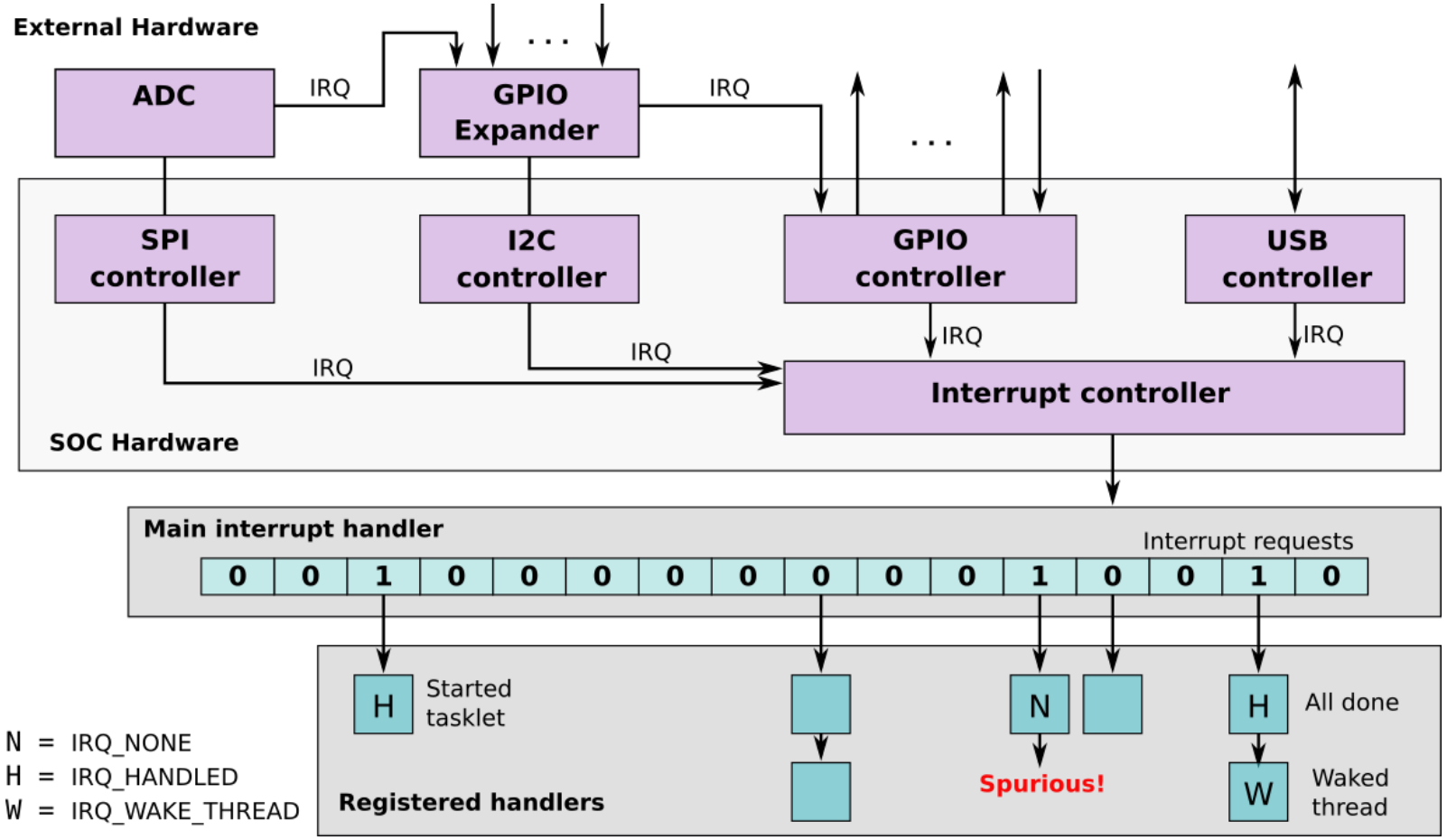
- void **kthread_init_worker**(struct kthread_worker *worker);
- bool **kthread_queue_work**(struct kthread_worker *worker,
  struct kthread_work *work);
- **kthread_init_work**(struct kthread_work *work, void (*fn)(struct kthread_work *work))

- void **kthread_flush_work**(struct kthread_work *work);
- void **kthread_flush_worker**(struct kthread_worker *worker);
- void **kthread_destroy_worker**(struct kthread_worker *worker);

From include/linux/kthread.h

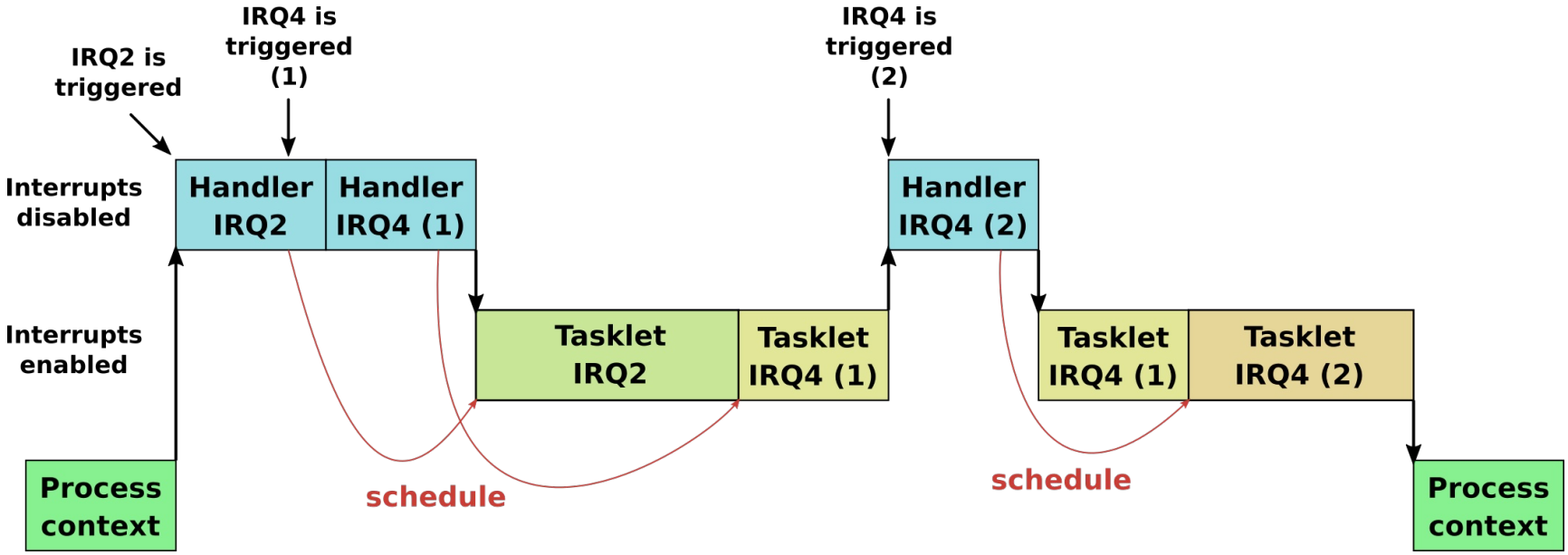
Interrupt Handling

Interrupt Handling: requests and handlers



Interrupt Handling: handlers and tasklets

Handler can schedule tasklet or wake threaded interrupt thread. Handler should return IRQ_NONE or IRQ_HANDLED. Tasklet can schedule work or wake thread.



Total do_softirq() execution time and restart count are limited:

```
#define MAX_SOFTIRQ_TIME    msecs_to_jiffies(2)
#define MAX_SOFTIRQ_RESTART 10
```

Interrupt Handling API

IRQ line request (register handlers) and free functions

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

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

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

/* Returns either IRQC_IS_HARDIRQ or IRQC_IS_NESTED */
int_request_any_context_irq(unsigned int irq, irq_handler_t handler,
                        unsigned long flags, const char *name, void *dev);

const void *free_irq(unsigned int irq, void *dev_id);
```

Probing the interrupt number (non-shared only). Do not use, see “IO resources”.

```
unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
```

Interrupt Handling API

IRQ flags (selected)

```
IRQF_SHARED       - allow sharing the irq among several devices
IRQF_PROBE_SHARED - set by callers when they expect sharing mismatches to occur
__IRQF_TIMER      - Flag to mark this interrupt as timer interrupt
IRQF_PERCPU       - Interrupt is per cpu
IRQF_NOBALANCING  - Flag to exclude this interrupt from irq balancing
IRQF_ONESHOT      - Interrupt is not reenabled after the hardirq handler finished.
                    Used by threaded interrupts which need to keep the
                    irq line disabled until the threaded handler has been run.
IRQF_NO_SUSPEND   - Do not disable this IRQ during suspend.
                    See Documentation/power/suspend-and-interrupts.txt
IRQF_NO_THREAD    - Interrupt cannot be threaded
```

IRQ types

```
IRQF_TRIGGER_RISING  - low-to-high transition
IRQF_TRIGGER_FALLING - high-to-low transition
IRQF_TRIGGER_HIGH    - high-level triggered
IRQF_TRIGGER_LOW     - low-level triggered
```

Interrupt Handling API: IRQ enable / disable

Single interrupt

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

All interrupts (they are really preprocessor macros)

```
void local_irq_enable();
void local_irq_disable();

void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

Interrupt Handling: threaded irq example (real code)

```
static irqreturn_t db1300_mmc_cd(int irq, void *ptr)
{
        disable_irq_nosync(irq);
        return IRQ_WAKE_THREAD;
}

static irqreturn_t db1300_mmc_cdfn(int irq, void *ptr)
{
        void (*mmc_cd)(struct mmc_host *, unsigned long);

        mmc_cd = symbol_get(mmc_detect_change);
        mmc_cd(ptr, msecs_to_jiffies(200));
        symbol_put(mmc_detect_change);

        msleep(100);    /* debounce */
        if (irq == DB1300_SD1_INSERT_INT)
                enable_irq(DB1300_SD1_EJECT_INT);
        else
                enable_irq(DB1300_SD1_INSERT_INT);

        return IRQ_HANDLED;
}
```

HardIRQ handler, atomic context
Interrupts are disabled

Threaded handler, non-atomic context

Interrupt Handling: BBB user button

Use gpio API to get interrupt number.
Do not forget to check return values and free resources.

```
button_irq = gpio_to_irq(button_gpio); // might return negative error code

rc = request_threaded_irq(button_irq, button_handler, button_thread,
                          IRQF_TRIGGER_FALLING,
                          "boot button irq test", &test);

free_irq(button_irq, &test);
```

Interrupt Handling: proc interface

BBB interrupts (example, not all lines are shown). See also /proc/stat

```
# cat /proc/interrupts
           CPU0
 16:    2212          INTC  68 Level     gp_timer
 18:       0          INTC   3 Level     arm-pmu
 26:       0          INTC  96 Level     44e07000.gpio
 28:       0          INTC  32 Level     481ac000.gpio
 30:     244          INTC  72 Level     OMAP UART0
 31:     154          INTC  70 Level     44e0b000.i2c
 33:      13          INTC  64 Level     mmc0
 34:      71          INTC  28 Level     mmc1
 45:    2028          INTC  41 Level     4a100000.ethernet
 46:     860          INTC  42 Level     4a100000.ethernet
 54:       1          INTC 111 Level     48310000.rng
 55:       0          INTC  18 Level     musb-hdrc.0
 56:       1          INTC  19 Level     musb-hdrc.1
 57:       0          INTC  17 Level     47400000.dma-controller
 58:       0 44e07000.gpio   6 Edge      48060000.mmc cd
 63:       0 481ac000.gpio   8 Edge      boot button irq test
IPI0:      0  CPU wakeup interrupts
IPI1:      0  Timer broadcast interrupts
IPI2:      0  Rescheduling interrupts
```

Interrupt Handling: threads

BBB (example, not all threads are shown)

```
 # ps
PID   USER           TIME  COMMAND
    1 root           0:01 init
    2 root           0:00 [kthreadd]
    5 root           0:00 [kworker/0:0-pm]
    9 root           0:00 [ksoftirqd/0]
   14 root           0:00 [kdevtmpfs]
   22 root           0:00 [edac-poller]
   23 root           0:00 [devfreq_wq]
   24 root           0:00 [watchdogd]
   28 root           0:00 [kswapd0]
   29 root           0:00 [nfsiod]
   42 root           0:00 [hwrng]
   45 root           0:00 [irq/58-48060000]
   46 root           0:00 [kworker/0:2-nfs]
   47 root           0:00 [ipv6_addrconf]
   66 root           0:00 -/bin/sh
   67 root           0:00 init
   68 root           0:00 init
   69 root           0:00 init
   82 root           0:00 [irq/63-boot_but]
   85 root           0:00 ps
```

References

- http://www.cs.fsu.edu/~cop4610t/lectures/project2/kthreads/kthreads.pdf
- Linux Device Drivers, Third Edition. Chapter 10: Interrupt Handling
- **Linux Kernel Interrupts and Handlers - Top and Bottom Halves**
- **Eliminating tasklets (LWN.net)**

Thanks!