Processes, scheduling and interrupts

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

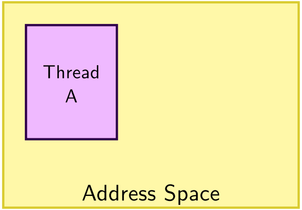
Processes and scheduling

Process, thread?

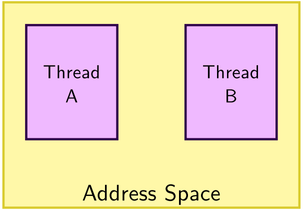
- Confusion about the terms *process*, *thread* and *task*
- In UNIX, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - A single thread, which is the only entity known by the scheduler.
- Additional threads can be created inside an existing process, using `pthread_create()`
  - They run in the same address space as the initial thread of the process
  - They start executing a function passed as argument to `pthread_create()`

Process, thread: kernel point of view

- In kernel space, each thread running in the system is represented by a structure of type `struct task_struct`
- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using `pthread_create()`



Process after fork()

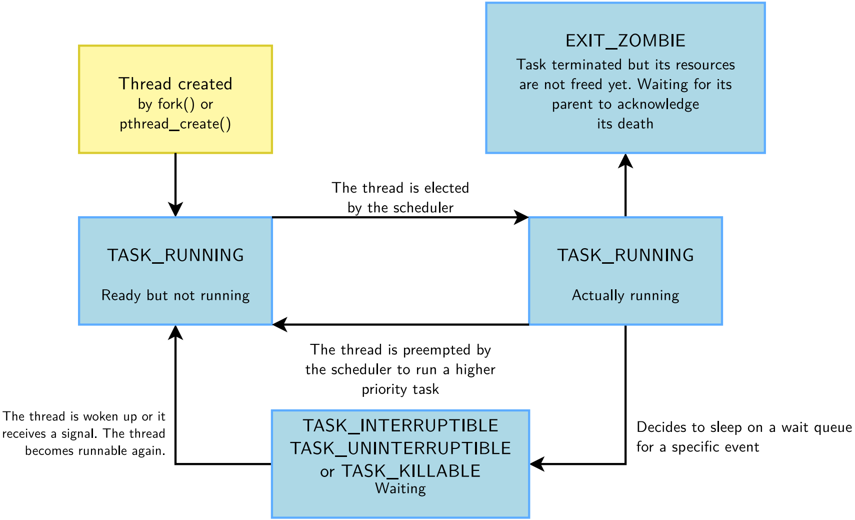


Same process after pthread_create()

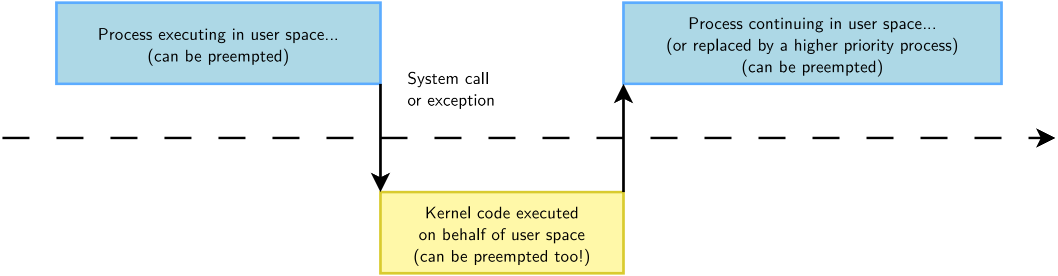
Relation between execution mode, address space and context

- When speaking about *process* and *thread*, these concepts need to be clarified:
  - *Mode* is the level of privilege allowing to perform some operations:
    - *Kernel Mode*: in this level CPU can perform any operation allowed by its architecture; any instruction, any I/O operation, any area of memory accessed.
    - *User Mode*: in this level, certain instructions are not permitted (especially those that could alter the global state of the machine), some memory areas cannot be accessed.
  - Linux splits its *address space* in *kernel space* and *user space*
    - *Kernel space* is reserved for code running in *Kernel Mode*.
    - *User space* is the place were applications execute (accessible from *Kernel Mode*).
  - *Context* represents the current state of an execution flow.
    - The *process context* can be seen as the content of the registers associated to this process: execution register, stack register...
    - The *interrupt context* replaces the *process context* when the interrupt handler is executed.

A thread life



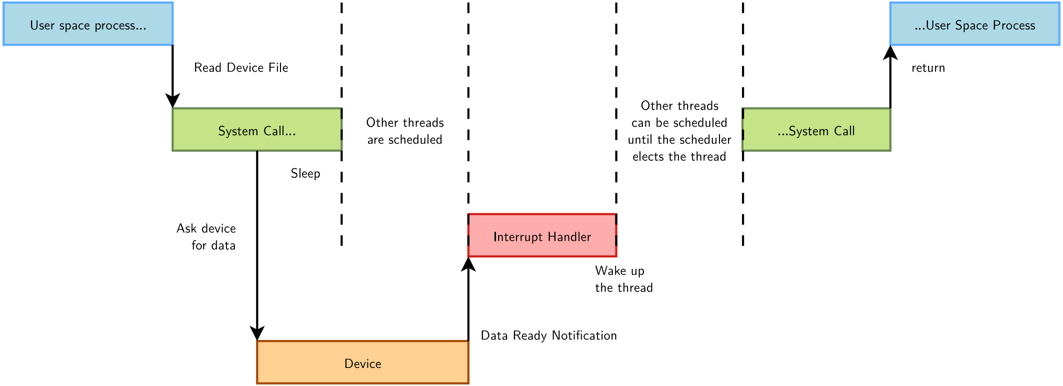
Execution of system calls



The execution of system calls takes place in the context of the thread requesting them.

Sleeping

Sleeping



Sleeping is needed when a process (user space or kernel space) is waiting for data.

How to sleep with a wait queue 1/3

- Must declare a wait queue, which will be used to store the list of threads waiting for an event
- Dynamic queue declaration:
  - Typically one queue per device managed by the driver
  - It's convenient to embed the wait queue inside a per-device data structure.
  - Example from drivers/net/ethernet/marvell/mvmdio.c:

```
struct orion_mdio_dev {
        ...
        wait_queue_head_t smi_busy_wait;
};
struct orion_mdio_dev *dev;
...
init_waitqueue_head(&dev->smi_busy_wait);
```

- Static queue declaration:
  - Using a global variable when a global resource is sufficient
  - `DECLARE_WAIT_QUEUE_HEAD(module_queue);`

How to sleep with a waitqueue 2/3

Several ways to make a kernel process sleep

- `void wait_event(queue, condition);`
  - Sleeps until the task is woken up **and** the given C expression is true. Caution: can't be interrupted (can't kill the user space process!)
- `int wait_event_killable(queue, condition);`
  - Can be interrupted, but only by a *fatal* signal (`SIGKILL`). Returns `-ERESTARTSYS` if interrupted.
- `int wait_event_interruptible(queue, condition);`
  - The most common variant
  - Can be interrupted by any signal. Returns `-ERESTARTSYS` if interrupted.

How to sleep with a waitqueue 3/3

- `int wait_event_timeout(queue, condition, timeout);`
  - Also stops sleeping when the task is woken up **or** the timeout expired (a timer is used).
  - Returns `0` if the timeout elapsed, non-zero if the condition was met.
- `int wait_event_interruptible_timeout(queue, condition, timeout);`
  - Same as above, interruptible.
  - Returns `0` if the timeout elapsed, `-ERESTARTSYS` if interrupted, positive value if the condition was met.

How to sleep with a waitqueue - Example

```
sig = wait_event_interruptible(ibmvtpm->wq,
                               !ibmvtpm->tpm_processing_cmd);
if (sig)
        return -EINTR;
```

From drivers/char/tpm/tpm_ibmvtpm.c

Waking up!

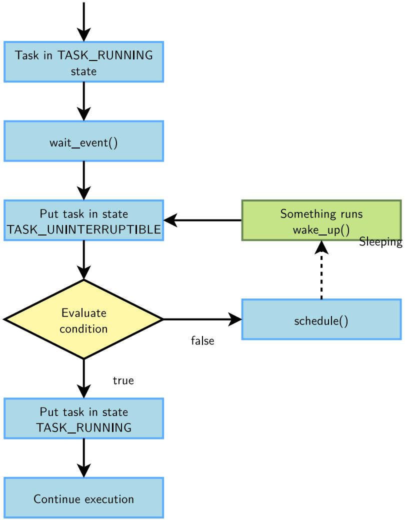
Typically done by interrupt handlers when data sleeping processes are waiting for become available.

- `wake_up(&queue);`
  - Wakes up all processes in the wait queue
- `wake_up_interruptible(&queue);`
  - Wakes up all processes waiting in an interruptible sleep on the given queue

Exclusive vs. non-exclusive

- wait_event_interruptible() puts a task in a non-exclusive wait.
  - All non-exclusive tasks are woken up by wake_up() / wake_up_interruptible()
- wait_event_interruptible_exclusive() puts a task in an exclusive wait.
  - wake_up() / wake_up_interruptible() wakes up all non-exclusive tasks and only one exclusive task
  - wake_up_all() / wake_up_interruptible_all() wakes up all non-exclusive and all exclusive tasks
- Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to “consume” the event.
- Non-exclusive sleeps are useful when the event can “benefit” to multiple tasks.

Sleeping and waking up - Implementation



The scheduler doesn't keep evaluating the sleeping condition!

- `wait_event(queue, cond);`
  The process is put in the `TASK_UNINTERRUPTIBLE` state.
- `wake_up(&queue);`
  All processes waiting in `queue` are woken up, so they get scheduled later and have the opportunity to evaluate the condition again and go back to sleep if it is not met.

See `include/linux/wait.h` for implementation details.

How to sleep with completions 1/2

- Use `wait_for_completion()` when no particular condition must be enforced at the time of the wake-up
  - Leverages the power of wait queues
  - Simplifies its use
  - Highly efficient using low level scheduler facilities
- Preparation of the completion structure:
  - Static declaration and initialization:

    `DECLARE_COMPLETION(setup_done);`
  - Dynamic declaration:

    `init_completion(&object->setup_done);`
  - The completion object should get a meaningful name (eg. not just "done").
- Ready to be used by signal consumers and providers as soon as the completion object is initialized
- See `include/linux/completion.h` for the full API
- Internal documentation at `scheduler/completion`

How to sleep with completions 2/2

- Enter a wait state with
  `void wait_for_completion(struct completion *done)`
  - All `wait_event()` flavors are also supported, such as: `wait_for_completion_timeout()`, `wait_for_completion_interruptible{,_timeout}()`, `wait_for_completion_killable{,_timeout}()`, etc
- Wake up consumers with
  `void complete(struct completion *done)`
  - Several calls to `complete()` are valid, they will wake up the same number of threads waiting on this object (acts as a FIFO).
  - A single `complete_all()` call would wake up all present and future threads waiting on this completion object
- Reset the counter with
  `void reinit_completion(struct completion *done)`
  - Resets the number of "done" completions still pending
  - Mind not to call `init_completion()` twice, which could confuse the enqueued tasks

Waiting when there is no interrupt

- When there is no interrupt mechanism tied to a particular hardware state, it is tempting to implement a custom busy-wait loop.
  - Spoiler alert: this is highly discouraged!
- For long lasting pauses, rely on helpers which leverage the system clock
  - `wait_event()` helpers are (also) very useful outside of interruption situations
  - Release the CPU with `schedule()`
- For shorter pauses, use helpers which implement software loops
  - `msleep()`/`msleep_interruptible()` put the process in sleep for a given amount of milliseconds
  - `udelay()`/`udelay_range()` waste CPU cycles in order to save a couple of context switches for a sub-millisecond period
  - `cpu_relax()` does nothing, but may be used as a way to not being optimized out by the compiler when busy looping for very short periods

Waiting when hardware is involved

- When hardware is involved in the waiting process
  - but there is no interrupt available
  - or because a context switch would be too expensive
- Specific polling I/O accessors may be used:
  - Exhaustive list in include/linux/iopoll.h

    ```
    int read[bwlq]_poll[_timeout[_atomic]](addr, val, cond,
                                           delay_us, timeout_us)
    ```

    - addr: I/O memory location
    - val: Content of the register pointed with
    - cond: Boolean condition based on val
    - delay_us: Polling delay between reads
    - timeout_us: Timeout delay after which the operation fails and returns -ETIMEDOUT
  - _atomic variant uses udelay() instead of usleep().

Sleeping summary

	Wait queue	**Completion**	**msleep()**	**udelay()**
Use in interrupt	Wake-up only	Wake-up only	NO!	Keep it short
Schedule latency	Scheduled	Scheduled	Scheduled	No

Interrupt Management

Registering an interrupt handler 1/2

The *managed* API is recommended:

```
int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
                     unsigned long irq_flags, const char *devname, void *dev_id);
```

- `device` for automatic freeing at device or module release time.
- `irq` is the requested IRQ channel. For platform devices, use `platform_get_irq()` to retrieve the interrupt number.
- `handler` is a pointer to the IRQ handler function
- `irq_flags` are option masks (see next slide)
- `devname` is the registered name (for `/proc/interrupts`). For platform drivers, good idea to use `pdev->name` which allows to distinguish devices managed by the same driver (example: `44e0b000.i2c`).
- `dev_id` is an opaque pointer. It can typically be used to pass a pointer to a per-device data structure. It cannot be `NULL` as it is used as an identifier for freeing interrupts on a shared line.

Releasing an interrupt handler

```
void devm_free_irq(struct device *dev, unsigned int irq, void *dev_id);
```

- Explicitly release an interrupt handler. Done automatically in normal situations.

Defined in `include/linux/interrupt.h`

Registering an interrupt handler 2/2

Here are the most frequent `irq_flags` bit values in drivers (can be combined):

- `IRQF_SHARED`: interrupt channel can be shared by several devices.
  - When an interrupt is received, all the interrupt handlers registered on the same interrupt line are called.
  - This requires a hardware status register telling whether an IRQ was raised or not.
- `IRQF_ONESHOT`: for use by threaded interrupts (see next slides). Keeping the interrupt line disabled until the thread function has run.

Interrupt handler constraints

- No guarantee in which address space the system will be in when the interrupt occurs: can't transfer data to and from user space.
- Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- Interrupt handlers are run with all interrupts disabled on the local CPU (see https://lwn.net/Articles/380931). Therefore, they have to complete their job quickly enough, to avoiding blocking interrupts for too long.

/proc/interrupts on Raspberry Pi 2 (ARM, Linux 4.19)

```
           CPU0       CPU1       CPU2       CPU3
 17:    1005317          0          0          0  ARMCTRL-level   1 Edge      3f00b880.mailbox
 18:         36          0          0          0  ARMCTRL-level   2 Edge      VCHIQ doorbell
 40:          0          0          0          0  ARMCTRL-level  48 Edge      bcm2708_fb DMA
 42:     427715          0          0          0  ARMCTRL-level  50 Edge      DMA IRQ
 56:  478426356          0          0          0  ARMCTRL-level  64 Edge      dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 80:     411468          0          0          0  ARMCTRL-level  88 Edge      mmc0
 81:        502          0          0          0  ARMCTRL-level  89 Edge      uart-pl011
161:          0          0          0          0  bcm2836-timer   0 Edge      arch_timer
162:   10963772    6378711   16583353    6406625  bcm2836-timer   1 Edge      arch_timer
165:          0          0          0          0  bcm2836-pmu     9 Edge      arm-pmu
FIQ:                                              usb_fiq
IPI0:         0          0          0          0  CPU wakeup interrupts
IPI1:         0          0          0          0  Timer broadcast interrupts
IPI2:   2625198    4404191    7634127    3993714  Rescheduling interrupts
IPI3:      3140      56405      49483      59648  Function call interrupts
IPI4:         0          0          0          0  CPU stop interrupts
IPI5:   2167923     477097    5350168     412699  IRQ work interrupts
IPI6:         0          0          0          0  completion interrupts
Err:          0
```

Note: interrupt numbers shown on the left-most column are virtual numbers when the Device Tree is used. The physical interrupt numbers can be found in `/sys/kernel/debug/irq/irqs/<nr>` files when `CONFIG_GENERIC_IRQ_DEBUGFS=y`.

Interrupt handler prototype

- `irqreturn_t foo_interrupt(int irq, void *dev_id)`
  - `irq`, the IRQ number
  - `dev_id`, the per-device pointer that was passed to `devm_request_irq()`
- Return value
  - `IRQ_HANDLED`: recognized and handled interrupt
  - `IRQ_NONE`: used by the kernel to detect spurious interrupts, and disable the interrupt line if none of the interrupt handlers has handled the interrupt.
  - `IRQ_WAKE_THREAD`: handler requests to wake the handler thread (see next slides)

Typical interrupt handler's job

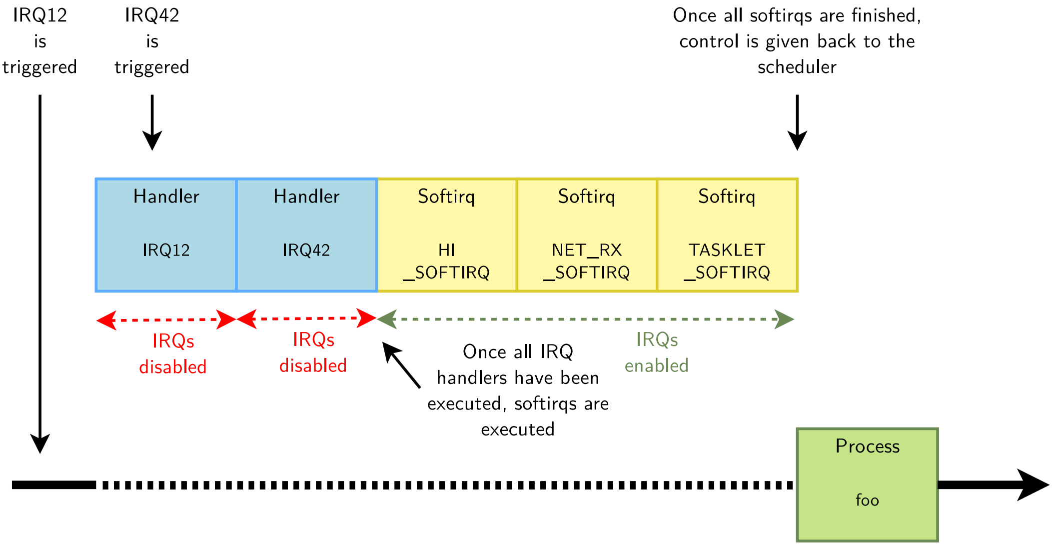
- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again)
- Read/write data from/to the device
- Wake up any process waiting for such data, typically on a per-device wait queue: `wake_up_interruptible(&device_queue);`

Top half and bottom half processing

Splitting the execution of interrupt handlers in 2 parts

- Top half
  - This is the real interrupt handler, which should complete as quickly as possible since all interrupts are disabled. It takes the data out of the device and if substantial post-processing is needed, schedule a bottom half to handle it.
- Bottom half
  - Is the general Linux name for various mechanisms which allow to postpone the handling of interrupt-related work. Implemented in Linux as softirqs, tasklets or workqueues.

Top half and bottom half diagram



Softirqs

- Softirqs are a form of bottom half processing
- The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
- They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
- The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by kernel subsystems (network, etc.)
- The list of softirqs is defined in `include/linux/interrupt.h`: `HI_SOFTIRQ`, `TIMER_SOFTIRQ`, `NET_TX_SOFTIRQ`, `NET_RX_SOFTIRQ`, `BLOCK_SOFTIRQ`, `IRQ_POLL_SOFTIRQ`, `TASKLET_SOFTIRQ`, `SCHED_SOFTIRQ`, `HRTIMER_SOFTIRQ`, `RCU_SOFTIRQ`
- `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` are used to execute tasklets

Example usage of softirqs - NAPI

NAPI = *New API*

- Interface in the Linux kernel used for interrupt mitigation in network drivers
- Principle: when the network traffic exceeds a given threshhold ("budget"), disable network interrupts and consume incoming packets through a polling function, instead of processing each new packet with an interrupt.
- This reduces overhead due to interrupts and yields better network throughput.
- The polling function is run by `napi_schedule()`, which uses `NET_RX_SOFTIRQ`.
- See https://en.wikipedia.org/wiki/New_API for details
- See also our commented network driver on https://bootlin.com/pub/drivers/r6040-network-driver-with-comments.c

Tasklets

- Tasklets are executed within the `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time.
- Tasklets are typically created with the `tasklet_init()` function, when your driver manages multiple devices, otherwise statically with `DECLARE_TASKLET()`. A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.
- The interrupt handler can schedule tasklet execution with:
  - `tasklet_schedule()` to get it executed in `TASKLET_SOFTIRQ`
  - `tasklet_hi_schedule()` to get it executed in `HI_SOFTIRQ` (highest priority)

Tasklet example: drivers/crypto/atmel-sha.c 1/2

```
/* The tasklet function */
static void atmel_sha_done_task(unsigned long data)
{
        struct atmel_sha_dev *dd = (struct atmel_sha_dev *)data;
        [...]
}

/* Probe function: registering the tasklet */
static int atmel_sha_probe(struct platform_device *pdev)
{
        struct atmel_sha_dev *sha_dd; /* Per device structure */
        [...]
        platform_set_drvdata(pdev, sha_dd);
        [...]
        tasklet_init(&sha_dd->done_task, atmel_sha_done_task,
                     (unsigned long)sha_dd);
        [...]
        err = devm_request_irq(&pdev->dev, sha_dd->irq, atmel_sha_irq,
                               IRQF_SHARED, "atmel-sha", sha_dd);
        [...]
}
```

Tasklet example: drivers/crypto/atmel-sha.c 2/2

```
/* Remove function: removing the tasklet */
static int atmel_sha_remove(struct platform_device *pdev)
{
        static struct atmel_sha_dev *sha_dd;
        sha_dd = platform_get_drvdata(pdev);
        [...]
        tasklet_kill(&sha_dd->done_task);
        [...]
}

/* Interrupt handler: triggering execution of the tasklet */
static irqreturn_t atmel_sha_irq(int irq, void *dev_id)
{
        struct atmel_sha_dev *sha_dd = dev_id;
        [...]
        tasklet_schedule(&sha_dd->done_task);
        [...]
}
```

Workqueues

- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts. It can typically be used for background work which can be scheduled.
- Workqueues may be created by subsystems or drivers with `alloc_workqueue()`. The default queue can also be used.
- Functions registered to run in workqueues, called workers, are executed in thread context which means:
  - All interrupts are enabled
  - Sleeping is allowed
- A worker is usually allocated in a per-device structure, initialized and registered with `INIT_WORK()` and typically triggered with `queue_work()` when using a dedicated queue or `schedule_work()` when using the default queue
- The complete API is in `include/linux/workqueue.h`
- Example (`drivers/crypto/atmel-i2c.c`):

```
INIT_WORK(&work_data->work, atmel_i2c_work_handler);
schedule_work(&work_data->work);
```

Threaded interrupts

The kernel also supports threaded interrupts:

- The interrupt handler is executed inside a thread.
- Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs time to communicate with them.
- Allows to set a priority for the interrupt handler execution, which is useful for real-time usage of Linux

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

- `handler`, "hard IRQ" handler
- `thread_fn`, executed in a thread

Bottom half summary

	softirq	tasklet	threaded IRQ	workqueue
Context	Interrupt	Interrupt	Process	Process
Can sleep	NO!	NO!	Yes	Yes (care if shared)
Dedicated process	-	-	Dedicated	Dedicated or shared
Schedule latency	"immediate"	"immediate"	Scheduled	Scheduled

Kernel threads (kthread)

- The general mechanism behind threaded IRQ and workqueues (and other things)
- Use a `struct task_struct` like userspace threads, but without a userspace address space and run solely in kernel mode
- Creation:
  - Use `kthread_create()` followed by `wake_up_process()`
  - Or simpler: `kthread_run()`
- Can be bound to a specific CPU using `kthread_bind()` (or directly using `kthread_run_on_cpu()`)
- `kthread_stop()` and `kthread_should_stop()` similar to pco-synchro ;-)
- To exit the kthread: `return;` from its main function

Interrupt management summary

- Device driver
  - In the `probe()` function, for each device, use `devm_request_irq()` to register an interrupt handler for the device's interrupt channel.
- Interrupt handler
  - Called when an interrupt is raised.
  - Acknowledge the interrupt
  - If needed, schedule a per-device tasklet taking care of handling data.
  - Wake up processes waiting for the data on a per-device queue
- Device driver
  - In the `remove()` function, for each device, the interrupt handler is automatically unregistered.