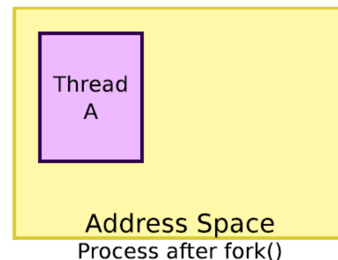
Rights to copy

- © Copyright 2004-2019, Bootlin
- **License: Creative Commons Attribution - Share Alike 3.0**
- https://creativecommons.org/licenses/by-sa/3.0/legalcode
- You are free:
  - to copy, distribute, display, and perform the work
  - to make derivative works
  - to make commercial use of the work
- Under the following conditions:
  - Attribution. You must give the original author credit.
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only
- under a license identical to this one.
  - For any reuse or distribution, you must make clear to others the license terms of this work.
  - Any of these conditions can be waived if you get permission from the copyright holder.
- Your fair use and other rights are in no way affected by the above.
- **Document sources:** https://git.bootlin.com/training-materials/

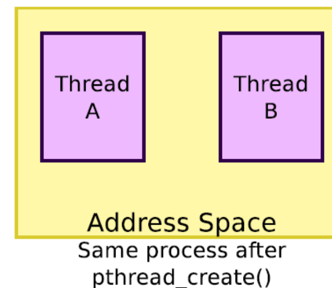
8. Task scheduling

Process, thread?

- Confusion about the terms process, thread and task
- A process is created using fork() and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the main() function.
  - Upon creation, a process contains one thread
- 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()
- The kernel represents each thread running in the system 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()

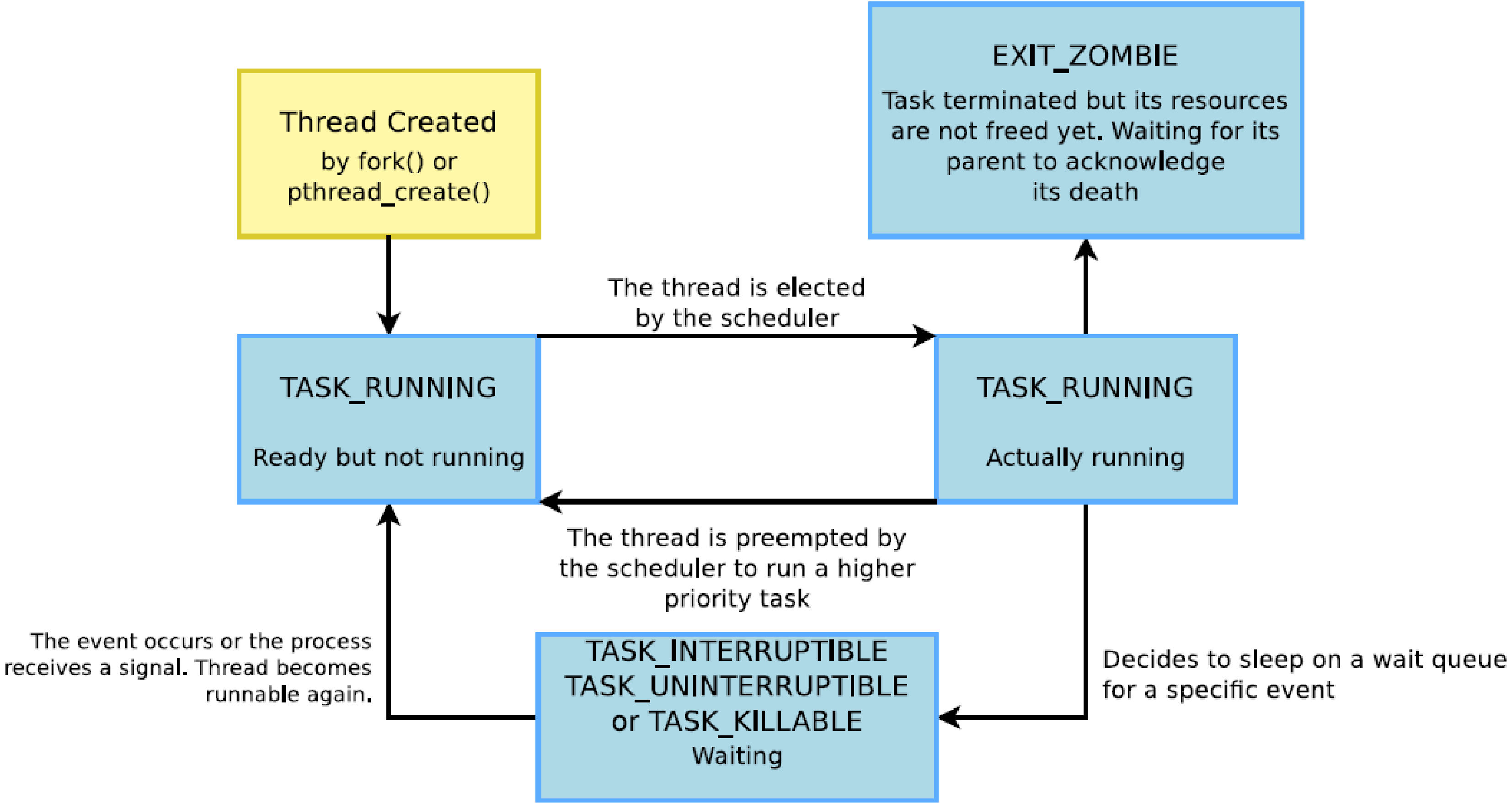


Process after fork()

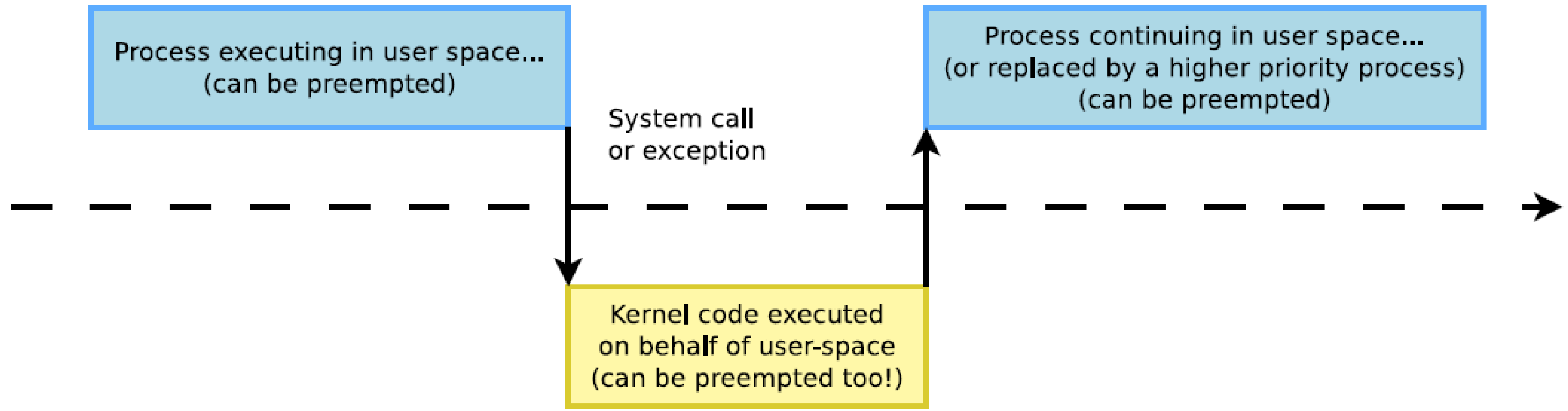


Same process after pthread_create()

A thread life



Execution of system calls



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

Task Schedulers and Load Balancing

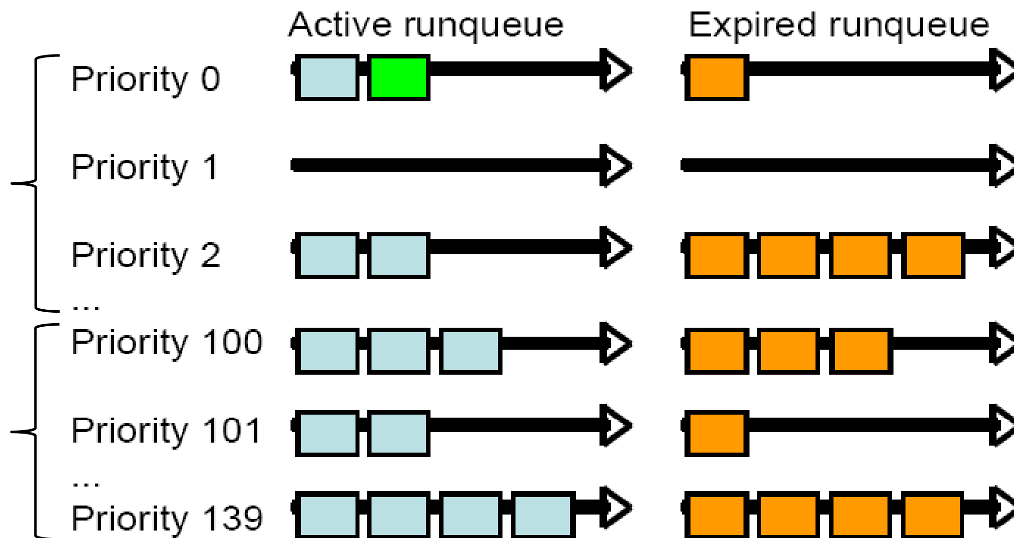
- The OS scheduler assigns tasks to cores by assessing a number of parameters, such as task priority, how much time the task has had, and how long it was since last run.
- Linux 2.6
  - O(1) scheduler
  - CFS (completely fair scheduler) since 2.6.23

The scheduler picks the first task in the active queue of the lowest priority level and lets it run. (Use Bitmap)

The active queue contains the tasks that are waiting to run.

real-time tasks

normal tasks (nice: -20~19)



The expired queue contains the tasks that have recently run and therefore must wait for the other queue to be empty.

140 priority levels (lower numbers indicate higher priority);
The top 40: normal user tasks, The lower 100: real-time tasks.

O(1) Scheduler

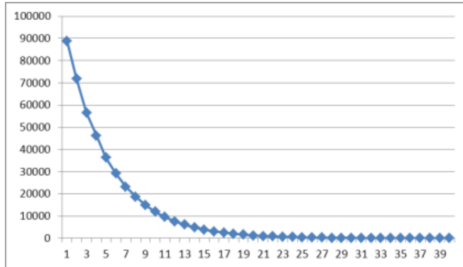
- As soon as the allocated execution time is used up the task is placed in the expired queue.
  - Time slice is allocated according to priority
  - If (priority<120) (140-priority)*20, else (140-priority)*5
  - Ex. Priority: 100 → 800ms, Priority: 120 → 100ms, Priority: 139 → 5ms
- Dynamic priority
  - max(100,min(static priority-bonus,139))
  - bonus(-5~5) is given based on the average sleep time
- When all tasks at the lowest priority level have run, the expired and active queues at that priority level switch places.
- If all the tasks are completed (for example, waiting for IO), the scheduler picks tasks from the second lowest priority level and so on.
- The tasks to be scheduled are always placed last in their priority levels queue.
- Keeps one such run-queue pair per core with individual locks on them.
- When a core needs to reschedule, it looks only in its own private run queue and can quickly decide which task should execute next.
- Load balancing among the cores occurs every 200 ms as a higher level scheduler analyzes the run queues to choose which task should move from one core to another.

CFS (completely fair scheduler)

- **To overcome some oddities of O(1) scheduler**
  - Time Slice based on priority
  - Different switching rate
    - Task 1 (prio=120), Task 2 (prio=120) → switching at every 100ms
    - Task 1 (prio=139), Task 2 (prio=139) → switching at every 5ms
  - Unfairness
    - Task 1 (prio=120), Task 2 (prio=121) → 100ms: 95ms
    - Task 1 (prio=138), Task 2 (prio=139) → 10ms: 5ms
  - Starvation of tasks in the expired queue
    - Due to some tasks in the active queue
  - No group scheduling (User A w/ 2 tasks and User B w/ 48 tasks)

CFS (completely fair scheduler)

- **Weight-based time slice**
  - Weight: based on priority



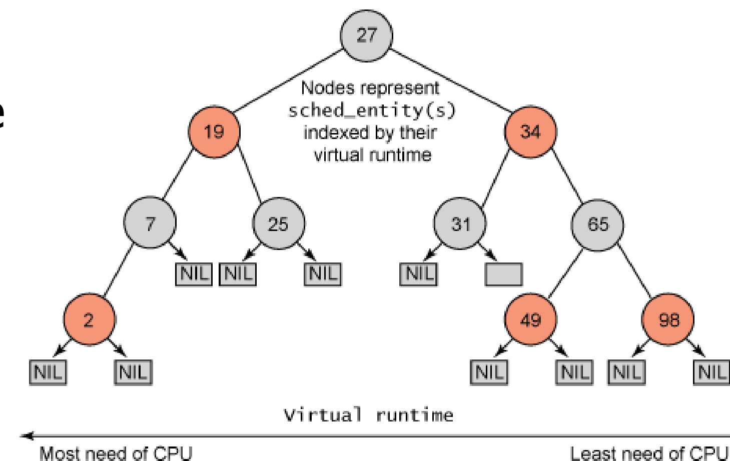

```
807 static const int prio_to_weight[40] = {
808  /* -20 */     88761,     71755,     56483,     46273,     36291,
809  /* -15 */     29154,     23254,     18705,     14949,     11916,
810  /* -10 */      9548,      7620,      6100,      4904,      3906,
811  /*  -5 */      3121,      2501,      1991,      1586,      1277,
812  /*   0 */      1024,       820,       655,       526,       423,
813  /*   5 */       335,       272,       215,       172,       137,
814  /*  10 */       110,        87,        70,        56,        45,
815  /*  15 */        36,        29,        23,        18,        15,
816 };
```

kernel/sched/sched.h

  - Time slice depends on weight, not priority
    - time slice = period *(se→load.weight/cfs_rq→load.weight)
    - period: the time slice the scheduler tries to execute all tasks (default 20ms)
    - se→load.weight: the weight of the schedulable entity. (prio_to_weight[ ])
    - cfs_rq→load.weight: the total weight of all entities under CFS runqueue.
    - Ex. Period = 100ms, task1 (prio=120), task2 (prio=121)
      - task1: 100*(1024/1024+820)=56ms
      - task2 : 100*(820/1024+820)=44ms
  - A schedulable entity can be a container, user, group (for user/group fair scheduling), or tasks.

CFS (completely fair scheduler)

- Uses virtual runtime to track the progress of each entity
- Virtual runtime is the weighted time slice given to every schedulable entity
  - vruntime = NICE_0_LOAD * (delta_exec/se→load.weight)
    - delta_exec: the amount of execution time of that task (equals to slice if there is no preemption from higher priority task).
    - NICE_0_LOAD: the unity value of the weight
  - increase as a task consumes time-slice
  - decrease as a task has a higher priority
- Uses a vruntime-based red-black tree for each CPU
  - Red-black tree is a self-balancing binary tree
  - All runnable tasks are sorted in the RB tree by se→vruntime
  - The leaf at the leftmost inside RB tree has the smallest value and most entitled to run at any given time.
- For interactivity optimization, CFS tunes the vruntime value that sleep for a period of time



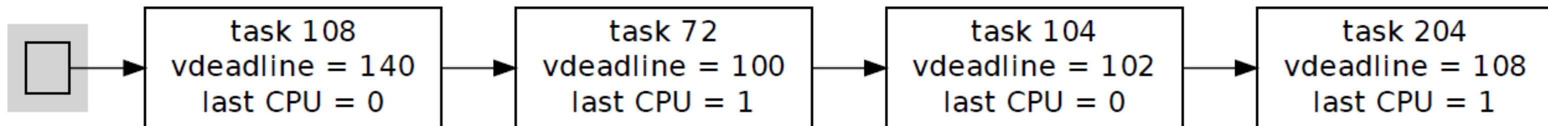
Load Balancing via Task Migration

- Linux/kernel/sched/fair.c

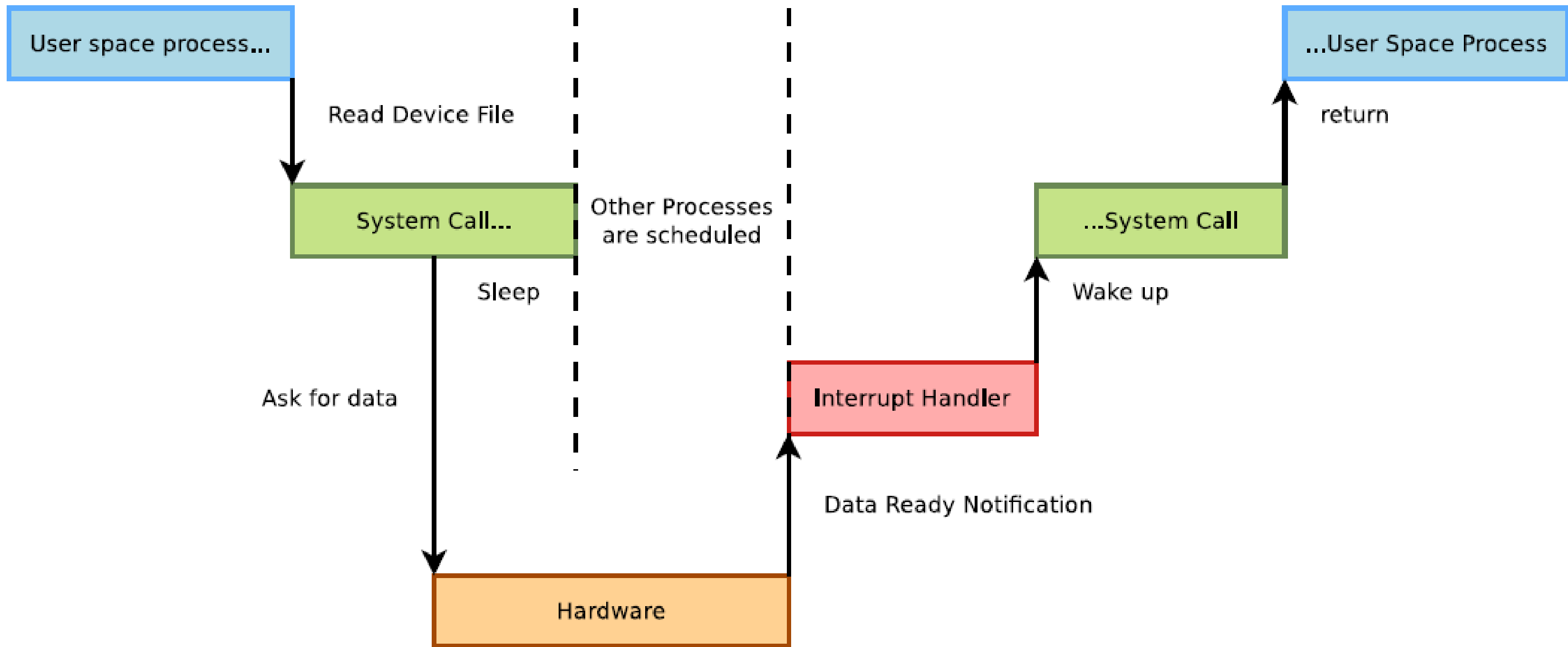
- scheduler_tick()
- trigger_load_balance()
- run_rebalance_domains() ; softirq(SCHED_SOFTIRQ)
- rebalance_domains()
- load_balance();
  - group = find_busiest_group();
    - calculate_imbalance();
  - busiest = find_busiest_queue();
  - move_tasks(); busiest → this_rq

BFS (Brain Fuck Scheduler)

- **O(1) and CFS maintain separate data structures for each CPU**
  - Reduce lock contention but requires explicit load balancing to evenly spread tasks across processors
- **BFS**
  - Only one system-wide runqueue containing all non-running tasks, O(n) scan over the entire queue.
  - earliest effective virtual deadline first policy
    - Tasks with higher priority are given earlier virtual deadlines
    - Gives incentive for a task to stay with its cache
  - tasks can be quickly scheduled on a different CPU that has become idle, often providing lower latency.
  - not in the mainline Linux kernel



Sleeping



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

How to sleep 1/2

- 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);

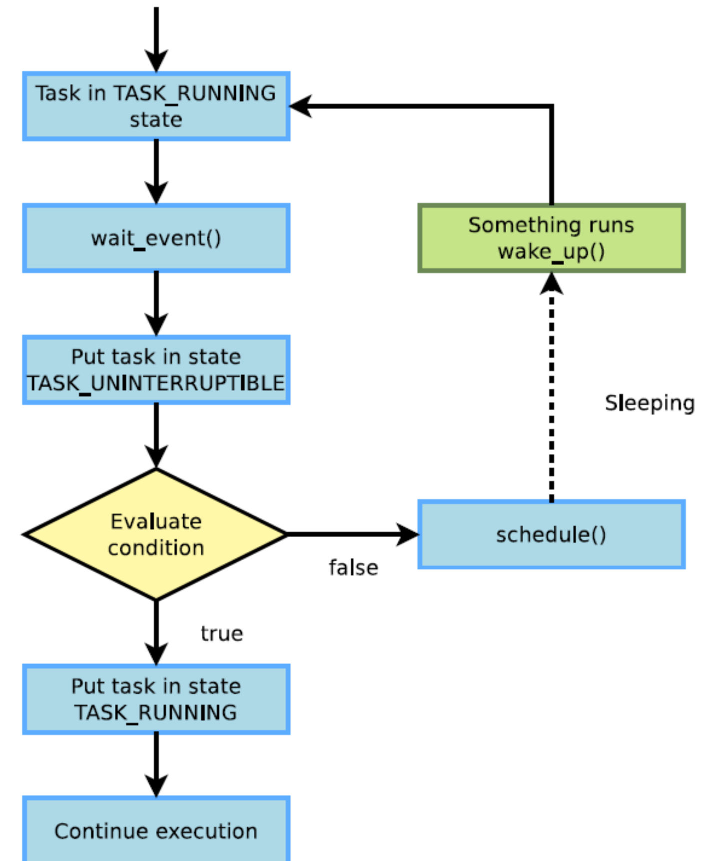
How to sleep 2/2

- 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 -ERESTARSYS if interrupted.
  - int wait_event_interruptible(queue, condition);
    - Can be interrupted by any signal.
    - Returns -ERESTARTSYS if interrupted.
  - int wait_event_timeout(queue, condition, timeout);
    - Also stops sleeping when the task is woken up and the timeout expired.
    - 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.

```
ret = wait_event_interruptible (sonypi_device.fifo_proc_list,
kfifo_len(sonypi_device.fifo) != 0);
if (ret) return ret;
```

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
- The scheduler doesn't keep evaluating the sleeping condition!
  - wait_event(queue, condition);
    - 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.



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.
  - prevent the thundering herd problem
- Non-exclusive sleeps are useful when the event can “benefit” to multiple tasks.

Interrupt Management

- Registering an interrupt handler
- 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
  - irq_flags are option masks
    - IRQF_SHARED: The interrupt channel can be shared by several devices. Requires a hardware status register telling whether an IRQ was raised or not.
    - IRQF_SAMPLE_RANDOM: Use the IRQ arrival time to feed the kernel random number generator.
  - devname is the registered name (for /proc/interrupts)
  - dev_id is a pointer to some data. It cannot be NULL as it is used as an identifier for free_irq() when using shared IRQs.

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

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.
  - Need to allocate memory with GFP_ATOMIC.
- Interrupt handlers are run with all interrupts disabled (since 2.6.36).
  - Have to complete their job quickly enough, to avoiding blocking interrupts for too long.

/proc/interrupts on Raspberry Pi 2 (ARM, Linux 4.14)

```
           CPU0       CPU1       CPU2       CPU3
 16:          0          0          0          0  bcm2836-timer   0 Edge      arch_timer
 17:   34723454   46066453   21374961   21330046  bcm2836-timer   1 Edge      arch_timer
 21:          0          0          0          0  bcm2836-pmu   9 Edge      arm-pmu
 23:    2039429          0          0          0  ARMCTRL-level   1 Edge      3f00b880.mailbox
 24:          2          0          0          0  ARMCTRL-level   2 Edge      VCHIQ doorbell
 46:          0          0          0          0  ARMCTRL-level  48 Edge      bcm2708_fb dma
 48:          0          0          0          0  ARMCTRL-level  50 Edge      DMA IRQ
 50:     644766          0          0          0  ARMCTRL-level  52 Edge      DMA IRQ
 59:          0          0          0          0  ARMCTRL-level  61 Edge      bcm2835-auxirq
 62: 1157888875          0          0          0  ARMCTRL-level  64 Edge      dwc_otg, dwc_otg_pcd, ...
 86:     641384          0          0          0  ARMCTRL-level  88 Edge      mmc0
 87:          3          0          0          0  ARMCTRL-level  89 Edge      uart-pl011
FIQ:              usb_fiq
IPI0:          0          0          0          0  CPU wakeup interrupts
IPI1:          0          0          0          0  Timer broadcast interrupts
IPI2:    3648739   13019827    4881211    4703599  Rescheduling interrupts
IPI3:          5         10         11          8  Function call interrupts
IPI4:          0          0          0          0  CPU stop interrupts
IPI5:    7601406   13651564    2755152    2939328  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 real physical interrupt numbers are either shown as an additional column, or can be seen in /sys/kernel/debug/irq_domain_mapping.

Interrupt handler prototype

- irqreturn_t foo_interrupt(int irq, void *dev_id)
  - irq, the IRQ number
  - dev_id, the opaque pointer that was passed to devm_request_irq()
- Return value
  - IRQ_HANDLED: recognized and handled interrupt
  - IRQ_NONE: not on a device managed by the module. Useful to share interrupt channels and/or report spurious interrupts to the kernel.

Typical interrupt handler's job

- Acknowledge the interrupt to the device
  - otherwise no more interrupts will be generated, or the interrupt will keep ring over and over again
- Read/write data from/to the device
- Wake up any waiting process waiting for the completion of an operation, typically using wait queues
  - wake_up_interruptible(&device_queue);

Threaded interrupts

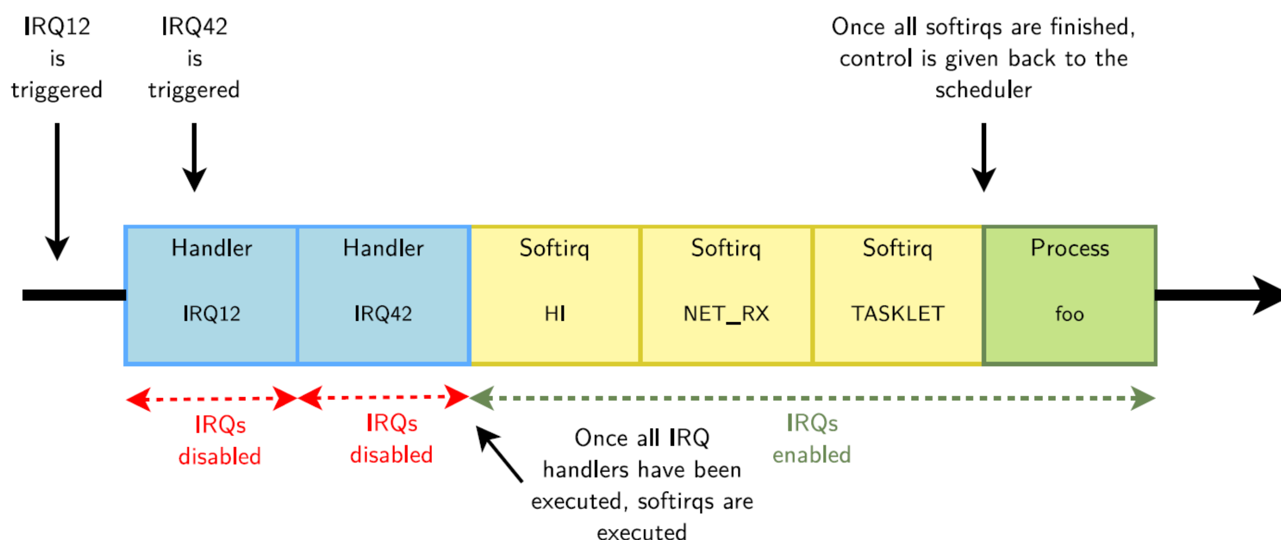
- Since 2.6.30, interrupt handler is executed inside a thread.
- Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs to communicate with them.
- Allows to set a priority for the interrupt handler execution, which is useful for real-time 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

Top half and bottom half processing

- Splitting the execution of interrupt handlers in 2 parts
- Top half
  - real interrupt handler
  - should complete as quickly as possible since all interrupts are disabled.
  - If possible, take the data out of the device and schedule a bottom half to handle it.
- Bottom half
  - 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.



Softirqs

- softirqs handlers are executed with all interrupts enabled, and can run simultaneously on multiple CPUs
- 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
  - not directly used by drivers, but by complete kernel subsystems (network, etc.)
- The list of softirqs is defined in
  - include/linux/interrupt.h
  - HI, TIMER, NET_TX, NET_RX, BLOCK, BLOCK_IOPOLL, TASKLET, SCHED, HRTIMER, RCU
  - HI and TASKLET softirqs are used to execute tasklets

Tasklets

- Tasklets are executed within the HI and TASKLET softirqs.
- executed with all interrupts enabled, but is guaranteed to execute on a single CPU at a time.
- A tasklet can be declared
  - statically with the DECLARE_TASKLET() macro
  - or dynamically with the tasklet_init() function.
  - 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 the execution of a tasklet with
  - tasklet_schedule() to get it executed in the TASKLET softirq
  - tasklet_hi_schedule() to get it executed in the HI softirq (higher priority)

Tasklet Example: drivers/crypto/atmel-sha.c

```
/* 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;
        [...]
        platform_set_drvdata(pdev, sha_dd);
        [...]
        tasklet_init(&sha_dd->done_task, atmel_sha_done_task,
                                        (unsigned long)sha_dd);
        [...]
}
```

```
/* 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

- general mechanism for deferring work.
- not limited in usage to handling interrupts.
- The function registered as workqueue is executed in a thread,
  - All interrupts are enabled
  - Sleeping is allowed
- A workqueue is registered with INIT_WORK() and typically triggered with queue_work()
- The complete API, in include/linux/workqueue.h provides many other possibilities (creating its own workqueue threads, etc.)

Interrupt management summary

- Device driver
  - When the device file is first opened, 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 tasklet taking care of handling data. Otherwise, wake up processes waiting for the data.
- Tasklet
  - Process the data
  - Wake up processes waiting for the data
- Device driver
  - When the device is no longer opened by any process, unregister the interrupt handler.