Linux Kernel Training. Lecture 6

Timers, Delays, Deferred Works

Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

April 10, 2020. GlobalLogic

Linux Kernel Time Sources

- Real-time clock (RTC)
  - Battery-backed HW clock;
  - Used to set and keep current date and time even when system is off;
- System Timers (Low Resolution): kernel/time/timer.c ;
  - Generate System ticks (100,250,1000 Hz);
  - Support System Time;
  - Tasks and Events scheduling;
- High Resolution Timers: kernel/time/hrtimer.c ;
  - Integrated into kernel mainline from 2.6.21;
  - Can support resolutions higher than 1 ms.
  - While S/W supports 1 ns resolution, normally rounded to the clock resolution of the specific platform.

Jiffies and HZ

- Jiffies
  - Until 2.6.21, jiffies was just a counter that was incremented every clock interrupt
  - Jiffies can wrap around depending on platform
    - 32 bits, 1000 HZ: about 50 days
    - 64 bits, 1000 HZ: about 600 million years
  - Jiffies_64:
    - On 64 bit machines, jiffies == jiffies_64;
    - On 32 bits, jiffies points to low-order 32 bits, jiffies_64 to high-order bits (be careful about atomicity!) =>

```
u64 get_jiffies_64(void);
```

- HZ
  - Determines how frequently the clock interrupt fires
  - Default is 1000 on x86, or 1 millisecond
  - Configurable at compile time or boot time
    Other typical values are 100 (10 ms) or 250 (4 ms)

- What's a good value for HZ?
  - Low values: less overhead
  - High values: better responsiveness

Kernel time structures

From include/uapi/linux/time(64).h

```
struct timespec(64) {
        __kernel_time_t   tv_sec;    /* seconds */
        long              tv_nsec;   /* nanoseconds */
};

struct timeval {
        __kernel_time_t      tv_sec;  /* seconds */
        __kernel_suseconds_t tv_usec; /* microseconds */
};
```

From include/uapi/linux/rtc.h

```
struct rtc_time {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;     /* unused */
    int tm_yday;     /* unused */
    int tm_isdst;    /* unused */
};
```

Conversion functions (`jiffies.h`)

```
unsigned long timespec_to_jiffies(struct timespec *value);
void          jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void          jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
unsigned int  jiffies_to_msecs(const unsigned long j);        /* the same for _usecs */
unsigned long msecs_to_jiffies(const unsigned int m);
```

Measuring Time Lapses

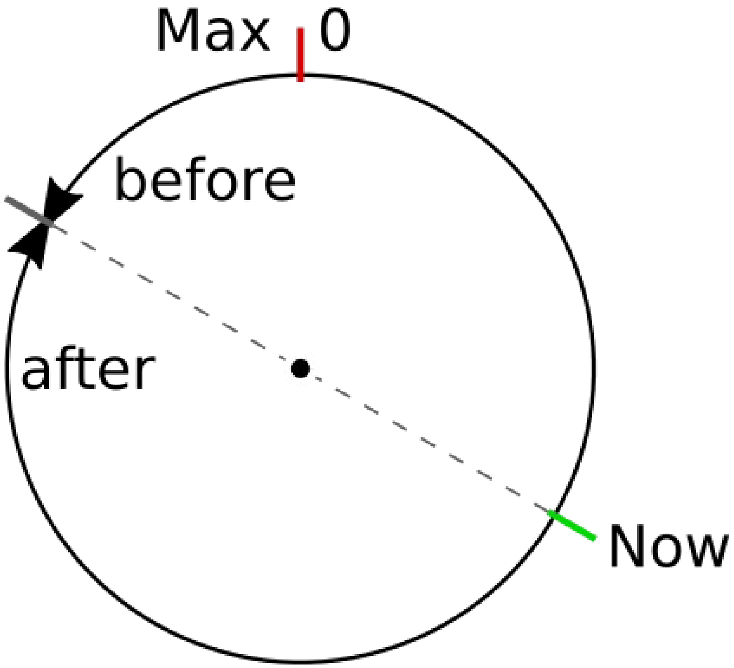
- Using jiffies

```
#include <linux/jiffies.h>
j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n*HZ/1000; /* n milliseconds */

bool time_after(unsigned long a, unsigned long b);
bool time_before(unsigned long a, unsigned long b);
bool time_after_eq(unsigned long a, unsigned long b);
bool time_before_eq(unsigned long a, unsigned long b);
```

- Similar for 64-bit

```
bool time_after64(u64 a, u64 b);
bool time_before64(u64 a, u64 b);
bool time_after_eq64(u64 a, u64 b);
bool time_before_eq64(u64 a, u64 b);
```



Processor Specific Registers

- x86:

  To access the timecounter, include <asm/msr.h> and use the following marcos

  ```
  /* read into two 32-bit variables */
  rdtsc(low32,high32);
  /* read low half into a 32-bit variable */
  rdtscl(low32);
  /* read into a 64-bit long long variable */
  rdtscll(var64);
  ```

  1-GHz CPU overflows the low half of the counter every 4.2 seconds

- Linux offers an architecture-independent function to access the architecture-specific cycle counter

  ```
  #include <linux/timex.h>
  cycles_t get_cycles(void);
  ```

  Returns 0 on platforms that have no cycle-counter register

Time Delays

- Busy-waiting

```
while (time_before(jiffies, j1))
        cpu_relax();

while (time_before(jiffies, j1))
        schedule();
```

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

- Non-busy waiting

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
void __sched usleep_range(unsigned long min, unsigned long max);
signed long __sched schedule_timeout(signed long timeout);
signed long __sched schedule_timeout_interruptible(signed long timeout);
signed long __sched schedule_timeout_uninterruptible(signed long timeout);
signed long __sched schedule_timeout_killable(signed long timeout);
```

Delays using WQ

See `linux/wait.h` (really they are macros):

```
DECLARE_WAIT_QUEUE_HEAD(name);
void wait_event(queue, condition);
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

Example:

```
#include <linux/wait.h>

wait_queue_head_t wait;

init_waitqueue_head(&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

Condition = 0 (no condition to wait for). Execution resumes when someone calls wake_up() or timeout expires;

Which function is best for me ?

- Documentation/timers/timers-howto.txt
  - "Is my code in an atomic context?"
  - This should be followed closely by "Does it really need to delay in atomic context?"

```
BUG: scheduling while atomic: swapper/1/0/0xffff0000
Modules linked in: tun libcomposite ipv6
[<c0014e4c>] (unwind_backtrace+0x0/0x11c) from [<c03a0720>] (__schedule_bug+0x48/0x5c)
[<c03a0720>] (__schedule_bug+0x48/0x5c) from [<c03a536c>] (__schedule+0x68/0x6e0)
[<c03a536c>] (__schedule+0x68/0x6e0) from [<c000ef08>] (cpu_idle+0xe4/0xfc)
```

- Are we doing bottom half or hardware interrupt processing? Are we in a softirq context? Interrupt context? See linux/preempt.h

```
#define in_irq()         (hardirq_count())
#define in_softirq()     (softirq_count())
#define in_interrupt()   (irq_count())
...
#define in_atomic()      ((preempt_count() != 0)
```

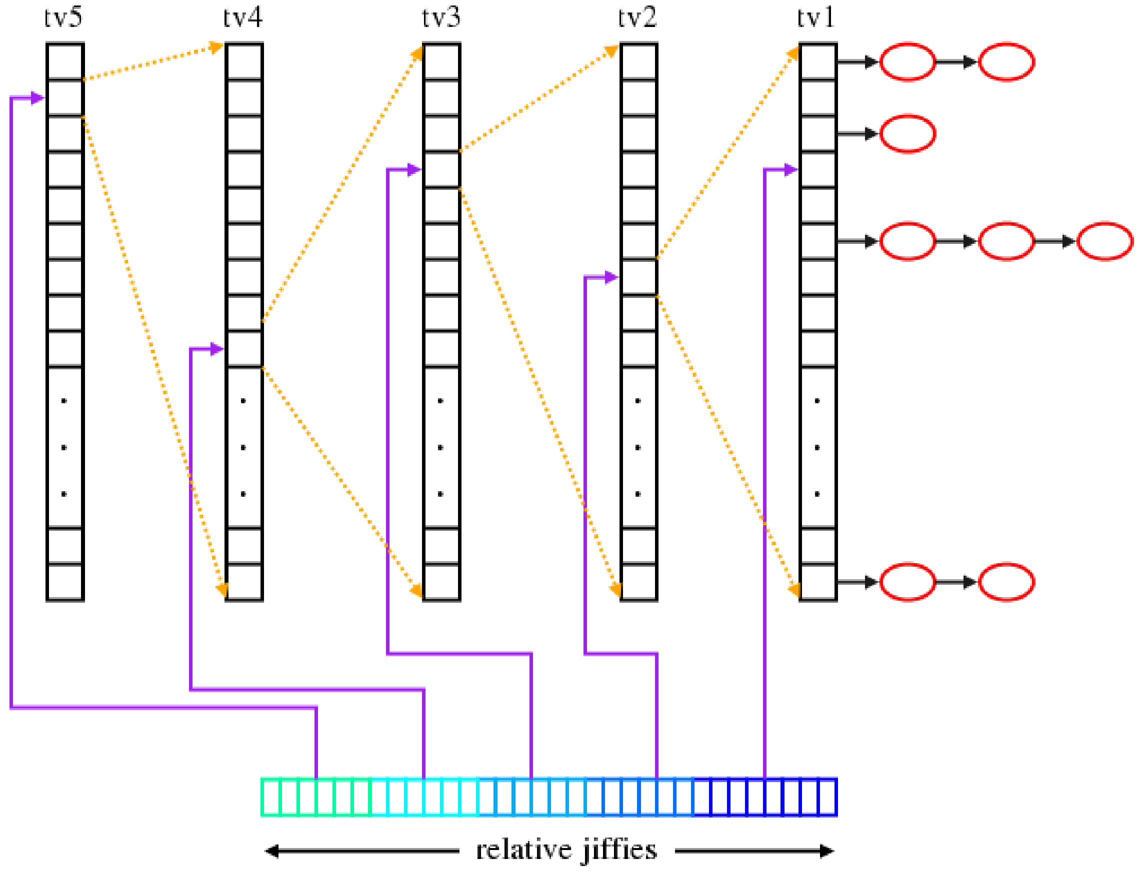
Contexts

- ATOMIC CONTEXT:
  You **must** use the *delay family of functions.

- NON-ATOMIC CONTEXT:
  You should use the *sleep[_range] family of functions.
  - SLEEPING FOR "A FEW" USECS (< ~10us?):
    Use udelay
  - SLEEPING FOR ~USECS OR SMALL MSECS (10us - 20ms):
    Use usleep_range
  - SLEEPING FOR LARGER MSECS (10ms+)
    Use msleep or possibly msleep_interruptible

Kernel Timers

Cascaded Timer Wheel



tv1 - containing a set of 256 (in most configurations) linked lists of upcoming timer events;

tv2 - set of 64 next level timers;

Cascading initiated after all timers of the given level expired;

Timer add and Timer delay complexity O(1)

Timer cascading complexity O(n)

Timer list structure

Up to kernel 4.14

`data` is a pointer to related structure (may be timer itself)

```
struct timer_list {
        struct hlist_node   entry;
        unsigned long       expires;
        void                (*function)(unsigned long);
        unsigned long       data;
        u32                 flags;
};

setup_timer(&mydev.timer, timer_callback, &mydev)

struct mydev *md = t->data;      /* in callback */
```

Since kernel 4.15

Use `container_of`-based macro `from_timer()`

```
struct timer_list {
        struct hlist_node   entry;
        unsigned long       expires;
        void                (*function)(struct timer_list *);
        u32                 flags;
};

timer_setup(&mydev.mytimer, timer_callback, TIMER_FLAGS);

struct mydev *md = from_timer(md, t, mytimer); /* in callback */
```

Kernel Timer API

- **Creation and manipulation**

```
void init_timer(struct timer_list *timer);
void init_timer_deferrable(struct timer_list *timer);
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
int del_timer_sync(struct timer_list *timer);
int mod_timer(struct timer_list *timer, unsigned long expires);
```

- **Example (drivers/pci/hotplug/cpqphp_ctrl.c, pre-4.15 style)**

```
init_timer(&p_slot->task_event);
p_slot->task_event.expires = jiffies + 5 * HZ;   /* 5 second delay */
p_slot->task_event.function = pushbutton_helper_thread;
p_slot->task_event.data = (u32) p_slot;
add_timer(&p_slot->task_event);
//
mod_timer(&my_timer, next_timeout);
// ----
del_timer(&p_slot->task_event);
```

High Resolution Timers

- Motivated by the observation of 2 types of timers:
    - Timeout functions, which we don't expect to actually happen (e.g., retransmission timer for packet loss). Have low resolution and are usually removed before expiration.
    - Timer functions, which we do expect to run. Have high resolution requirements and usually expire
- Original timer implementation is based on jiffies and thus depends on HZ.
    - Works well for timeouts, less so for timers.
    - Resolution no better than HZ (e.g., 1 millisecond)
- High resolution timers, introduced in 2.6.16, allow 1 nanosecond resolution

  Implemented in an red-black tree (rbtree)
- Insert, delete, search in O(log n) time

High Resolution Timers API

- #include <linux/ktime.h>

```
union ktime {
        s64 tv64;    // in nanoseconds
};
```

- Initialization of time variable (defined in include/linux/ktime.h)

```
ktime_t kt;
kt = ktime_set(long secs, long nanosecs);

ktime_t ktime_add(ktime_t kt1, ktime_t kt2);
ktime_t ktime_sub(ktime_t kt1, ktime_t kt2);  /* kt1 - kt2 */
ktime_t ktime_add_ns(ktime_t kt, u64 nanoseconds);
ktime_t timespec_to_ktime(struct timespec tspec);
ktime_t timeval_to_ktime(struct timeval tval);
struct timespec ktime_to_timespec(ktime_t kt);
struct timeval ktime_to_timeval(ktime_t kt);
clock_t ktime_to_clock_t(ktime_t kt);
u64 ktime_to_ns(ktime_t kt);
```

High Resolution Timers API

void hrtimer_init(struct hrtimer *timer, clockid_t which_clock);

1. CLOCK_MONOTONIC: a clock which is guaranteed always to move forward in time, but which does not reflect "wall clock time"
2. CLOCK_REALTIME which matches the current real-world time.

void hrtimer_rebase(struct hrtimer *timer, clockid_t new_clock);

- Callback function :

```
int  (*function)(void *);
```

  - HRTIMER_NORESTART
  - HRTIMER_RESTART
- Setting restart time:

```
u64 hrtimer_forward(struct hrtimer *timer, ktime_t now, ktime_t interval);
u64 hrtimer_forward_now(struct hrtimer *timer,ktime_t interval);
static inline u64 hrtimer_forward_now(struct hrtimer *timer, ktime_t interval)
{
        return hrtimer_forward(timer, timer->base->get_time(), interval);
}
```

High Resolution Timers API

- **int hrtimer_start(struct hrtimer *timer, ktime_t time, enum hrtimer_mode mode);**
  - HRTIMER_ABS
  - HRTIMER_REL
- **int hrtimer_cancel(struct hrtimer *timer);**
  - The return value will be zero if the timer was not active (meaning it had already expired, normally), or one if the timer was successfully canceled;
- **int hrtimer_try_to_cancel(struct hrtimer *timer);**
  - Returns -1 if the timer function is running;
- **void hrtimer_restart(struct hrtimer *timer)** - can restart cancelled timer;
- **ktime_t hrtimer_get_remaining(const struct hrtimer *timer);**
- **bool hrtimer_active(const struct hrtimer *timer);**
- **int hrtimer_get_res(clockid_t which_clock, struct timespec *tp);**

HRT Example

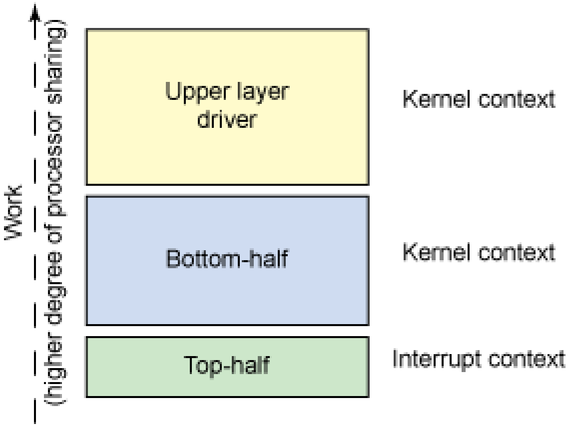
```
int init_module(void)
{
        ktime_t ktime;

        ktime = ktime_set(0, MS_TO_NS(delay_in_ms));
        hrtimer_init(&hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
        hr_timer.function = &my_hrtimer_callback;
        hrtimer_start( &hr_timer, ktime, HRTIMER_MODE_REL );
        return 0;
}

enum hrtimer_restart my_hrtimer_callback( struct hrtimer *timer)
{
          if (restart--) {
                    hrtimer_forward_now(timer, ns_to_ktime(MS_TO_NS(delay_in_ms)));
                    return HRTIMER_RESTART;
          }
          return HRTIMER_NORESTART;
}
```

Tasklets and Workqueues

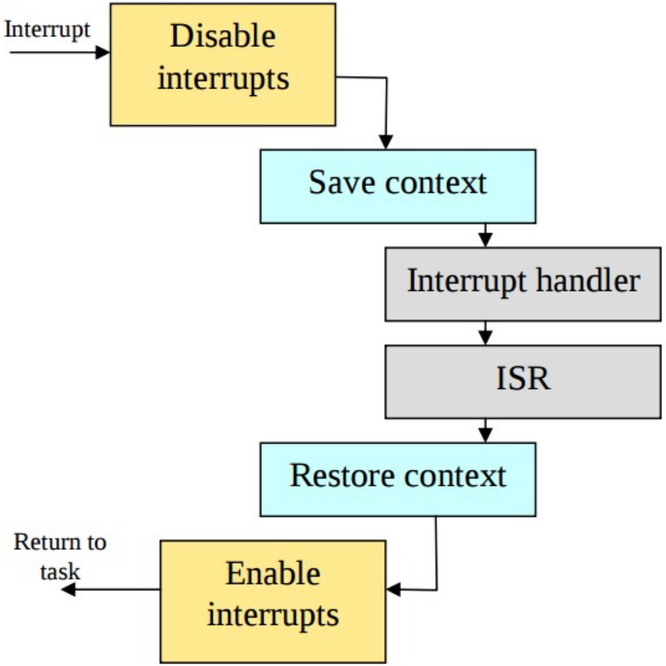
Top and Bottom half processing



Three types of Bottom halves

in the Linux kernel:

- softirqs;
- tasklets;
- workqueues;



Bottom Half	**Context**	**Inherent Serialization**
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queues	Process	None (scheduled as process context)

Softirqs

- Each processor has its own thread that is called **`ksoftirqd/n`** where the **`n`** is the number of the processor.
- Softirqs are determined statically at compile-time of the Linux kernel and the `open_softirq` function takes care of `softirq` initialization. The `open_softirq` function defined in the kernel/softirq.c:

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];

void open_softirq(int nr, void (*action)(struct softirq_action *))
{
        softirq_vec[nr].action = action;
}

void raise_softirq(unsigned int nr)
{
        unsigned long flags;

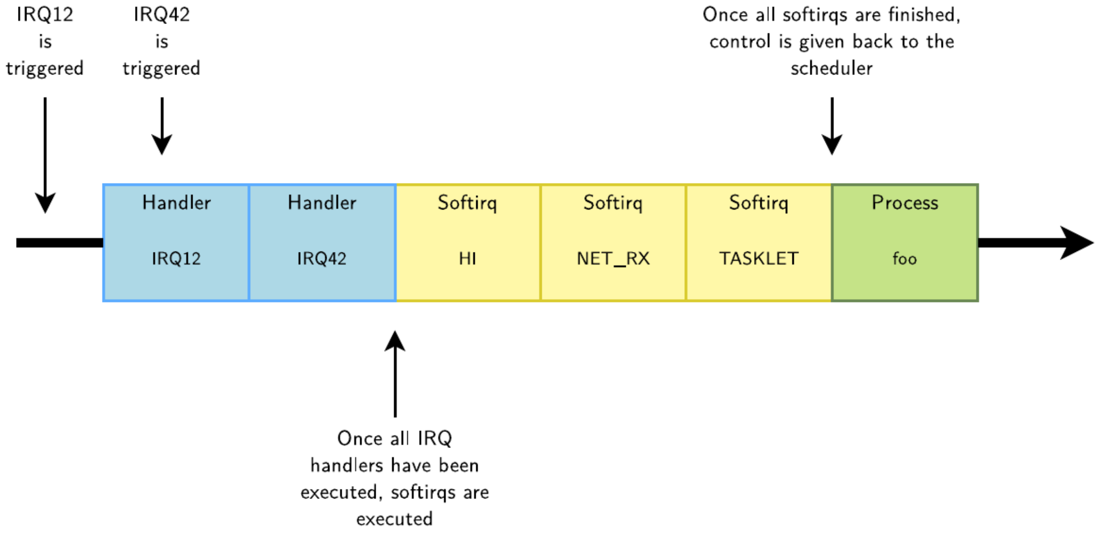
        local_irq_save(flags);
        raise_softirq_irqoff(nr);
        local_irq_restore(flags)
}

do_softirq();
```

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,
    NR_SOFTIRQS
};
```

Softirqs

- 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 complete kernel subsystems (network, etc.)
- The list of softirqs is defined in include/linux/interrupt.h:
- The HI and TASKLET softirqs are used to execute tasklets



Tasklets

- Executed with all interrupts enabled, but a given tasklet 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.
- 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;

  #include <linux/interrupt.h>

- ```
  struct tasklet_struct
  {
          struct tasklet_struct *next;// linked list
          unsigned long state; // scheduled or running? (for waiting)
          atomic_t count; // enabled(0) or disabled?
          void (*func)(unsigned long);// function pointer, i.e. bottom half
          unsigned long data; // argument for function
  };
  ```
- Stats: zero, TASKLET_STATE_SCHED, or TASKLET_STATE_RUN
  ```

Tasklets API

```
DECLARE_TASKLET( name, func, data );
DECLARE_TASKLET_DISABLED( name, func, data);
void tasklet_init( struct tasklet_struct *, void (*func)(unsigned long),
            unsigned long data );
void tasklet_disable_nosync( struct tasklet_struct * );
void tasklet_disable( struct tasklet_struct * );
void tasklet_enable( struct tasklet_struct * );
void tasklet_hi_enable( struct tasklet_struct * );
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
void tasklet_kill( struct tasklet_struct * ); /* will wait for its completion, and then kill it */
```

Tasklets

Example (LDD3 examples: jit.c, jit_tasklet_proc_show)

```
/* register the tasklet */
tasklet_init(&data->tlet, jit_tasklet_fn, (unsigned long)data);
data->hi = hi;
if (hi)
        tasklet_hi_schedule(&data->tlet);
else
        tasklet_schedule(&data->tlet);

/* wait for the buffer to fill */
wait_event_interruptible(data->wait, !data->loops);
```

Tasklets

```
void jit_tasklet_fn(unsigned long arg) {
          struct jit_data *data = (struct jit_data *) arg;
          unsigned long j = jiffies;

          data->buf += sprintf(data->buf, "%9li %3li %i %6i %i %s\n", j,
                               j - data->prevjiffies, in_interrupt() ? 1 : 0,
                               current->pid, smp_processor_id(), current->comm);

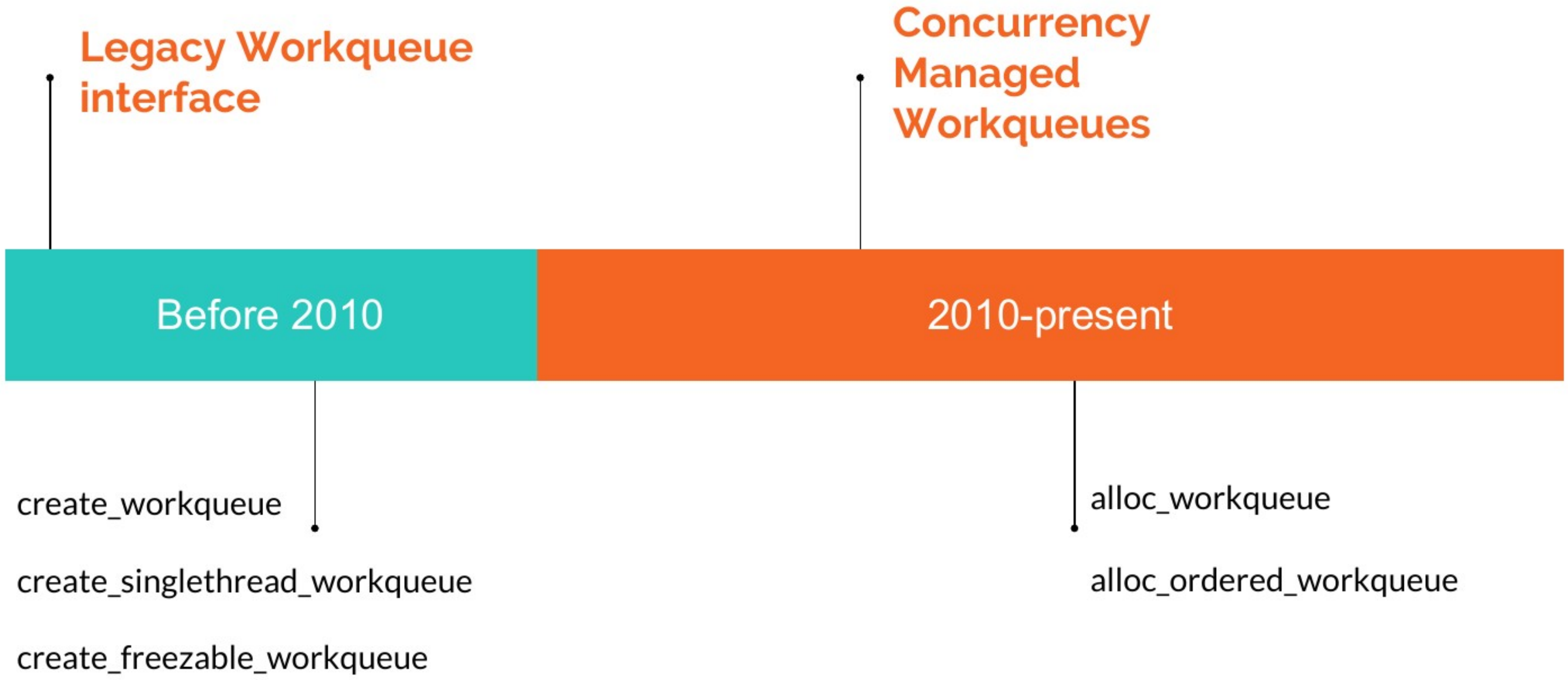
          if (--data->loops) {
                    data->prevjiffies = j;
                    if (data->hi)
                            tasklet_hi_schedule(&data->tlet);
                    else
                            tasklet_schedule(&data->tlet);
          } else {
                    wake_up_interruptible(&data->wait);
          }
}
```

Workqueues

- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts.
- The function registered as workqueue is executed in a thread, which means:
  - All interrupts are enabled;
  - Sleeping is allowed;
- A work can be registered on System (shared) WQ or declare it’s own;
- The complete API, in include/linux/workqueue.h provides many other possibilities (creating its own workqueue threads, etc.)

Workqueues

History



Worker pool, Worker and Work

```
struct worker_pool {
        spinlock_t              lock;           /* the pool lock */
        int                     cpu;            /* I: the associated cpu */
        int                     node;           /* I: the associated node ID */
        int                     id;             /* I: pool ID */
        unsigned int            flags;          /* X: flags */

        unsigned long           watchdog_ts;    /* L: watchdog timestamp */

        struct list_head        worklist;       /* L: list of pending works */
        int                     nr_workers;     /* L: total number of workers */

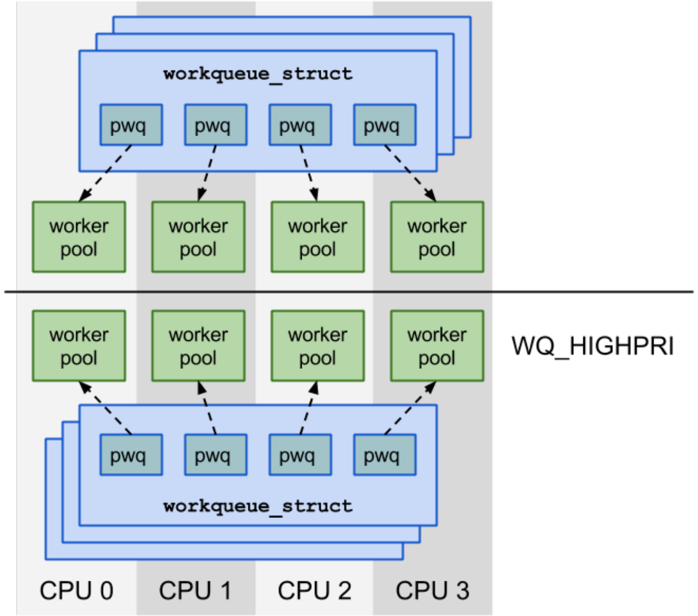
...
```

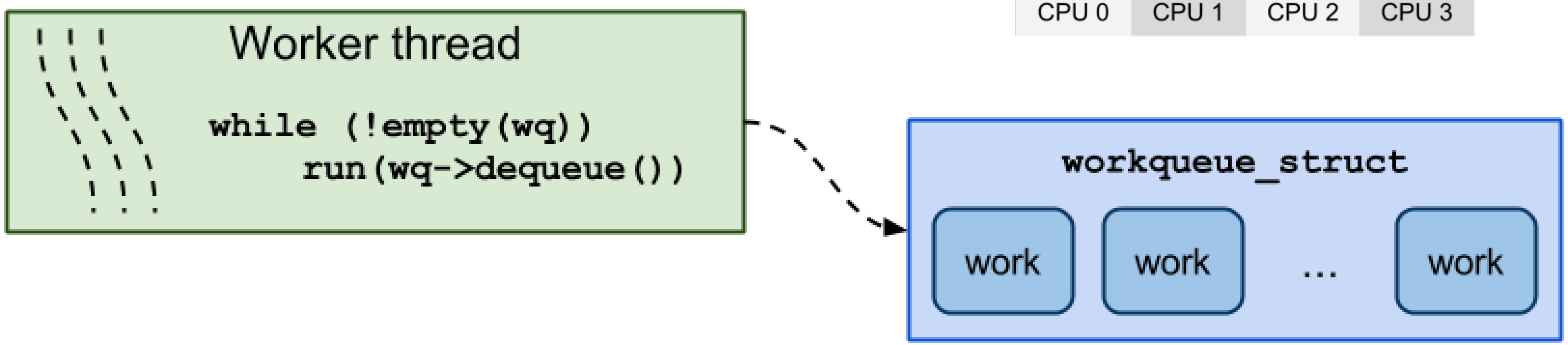
Worker pool, Worker and Work

```
struct worker_pool {
        spinlock_t          lock;
        int                 cpu;
        int                 node;
        int                 id;
        unsigned int        flags;

        unsigned long  watchdog_ts;

        struct list_head        worklist;
        int                 nr_workers;
```





WQ and Work

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
#ifdef CONFIG_LOCKDEP
        struct lockdep_map lockdep_map;
#endif
};

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;
};
```

Workqueue API

- Create and Destroy Workqueue
  - struct workqueue_struct ***create_workqueue**(name); - deprecated;

    #define alloc_workqueue(fmt, flags, max_active, args...)

    #define alloc_ordered_workqueue(fmt, flags, args...)
  - void **destroy_workqueue**(struct workqueue_struct *);
- Init work item
  - **INIT_WORK**(work, func);
  - **INIT_DELAYED_WORK**(work, func);
  - **INIT_DELAYED_WORK_DEFERRABLE**(work, func);
- Scheduling dedicated WQ
  - bool **queue_work**(struct workqueue_struct *wq, struct work_struct *work);
  - bool **queue_work_on**(int cpu, struct workqueue_struct *wq, struct work_struct *work);
  - bool **queue_delayed_work**(struct workqueue_struct *wq,

    struct delayed_work *dwork, unsigned long delay);
  - bool **queue_delayed_work_on**(int cpu, struct workqueue_struct *wq,

    struct delayed_work *dwork, unsigned long delay);

Workqueue API

- Scheduling shared WQ
  - bool schedule_work(struct work_struct *work);
  - bool schedule_work_on(int cpu, struct work_struct *work);
  - bool scheduled_delayed_work(struct delayed_work *dwork, unsigned long delay);
  - bool scheduled_delayed_work_on(int cpu, struct delayed_work *dwork, unsigned long delay);
- Flushing WQ
  - int flush_work(struct work_struct *work);
  - int flush_workqueue(struct workqueue_struct *wq);
  - void flush_scheduled_work(void);
- Cancelling WQ
  - int cancel_work_sync(struct work_struct *work);
  - int cancel_delayed_work_sync(struct delayed_work *dwork);

Workqueue API

- Checking pending Work
  - work_pending(work);
  - delayed_work_pending(work);
- WQ Flags
  - **WQ_UNBOUND** - Unbound to any specific CPU
  - **WQ_FREEZABLE** - Should handle system freezes
  - **WQ_MEM_RECLAIM** - Create separated “rescue” thread for memory reclaims
  - **WQ_HIGHPRI** - high priority WQ
  - **WQ_CPU_INTENSIVE** - runnable CPU intensive work items will not prevent other work items in the same worker pool from starting execution (useless with **WQ_UNBOUND**)
  - **max active** - determines the maximum number of execution contexts (workers) per CPU of the respective workqueue.

System wide WQs

- **system_wq** - System-wide multi-threaded workquque used by various variants of schedule work(). Do not queue long-running work items, since users expect a relatively short flush tim.
- **system_highpri_wq** - Similar to system wq but for work items which require WQ_HIGHPRI.
- **system_long_wq** - Similar to system wq but may host long running works. Queue flushing is expected to take relatively long.
- **system_unbound_wq** - Unbound workqueue. Workers are not bound to any specific CPU, not concurrency managed, and all queued works are executed immediately as long as max active limit is not reached and resources are available.
- **system_freezable_wq** - Equivalent to system wq but with WQ_FREEZABLE enabled.
- **power efficient wq** - Inclined towards saving power and converted into WQ_UNBOUND variants if WQ_POWER_EFFICIENT is set.
- **system_freezable_power_efficient_wq** - Combination of system freezable wq and power efficient wq.

Subsystem-related WQ might be available. For example, per-hw queues in 802.11:

- `void ieee80211_queue_work(struct ieee80211_hw *hw, struct work_struct *work);`
- `void ieee80211_queue_delayed_work(struct ieee80211_hw *hw, struct work_struct *work);`

WQ Scheduling

- Work items w0, w1, w2 are queued to a bound wq q0 on the same CPU. w0 burns CPU for 5ms then sleeps for 10ms then burns CPU for 5ms again before finishing. w1 and w2 burn CPU for 5ms then sleep for 10ms.

- **Original WQ**
- **TIME IN MSECS   EVENT**

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 starts and burns CPU
25	w1 sleeps
35	w1 wakes up and finishes
35	w2 starts and burns CPU
40	w2 sleeps
50	w2 wakes up and finishes

- **CMWQ with @max_active >= 3**
- **TIME IN MSECS   EVENT**

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 starts and burns CPU
10	w1 sleeps
10	w2 starts and burns CPU
15	w2 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
25	w2 wakes up and finishes

WQ Scheduling

- **@max_active == 2,**
- **TIME IN MSECS EVENT**

TIME IN MSECS	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 starts and burns CPU
10	w1 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
20	w2 starts and burns CPU
25	w2 sleeps
35	w2 wakes up and finishes

- **w1 and w2 are queued to a different wq q1 which has WQ_HIGHPRI set**

TIME IN MSECS	EVENT
0	w1 and w2 start and burn CPU
5	w1 sleeps
10	w2 sleeps
10	w0 starts and burns CPU
15	w0 sleeps
15	w1 wakes up and finishes
20	w2 wakes up and finishes
25	w0 wakes up and burns CPU
30	w0 finishes

References

- Linux Device Drivers, Third Edition. [Chapter 7: *Time, Delays, and Deferred Work*]
- LDD3 examples https://github.com/duxing2007/ldd3-examples-3.x
  - `jit.c` is used for some code examples in this presentation
- Documentation/timers/timers-howto.txt
- Documentation/timers/hrtimers.txt
- The high-resolution timer API (LWN.net)
- Deferrable functions, kernel tasklets, and work queues (developer.ibm.com)
- www.cs.columbia.edu/~nahum/w6998/lectures/timers.ppt

Thanks!