Interrupt Handler: Bottom Half

Xiaoguang Wang

UIC COMPUTER SCIENCE

Homework

hw3 grading is out

paper reading due yesterday (Feb/12th)

hw4 & hw5: kernel module

- use the ubuntu cloud image if you have difficulty to access a Linux machine
- Instructions on blackboard discussion (Environment Settings)

Recap: Interrupt controller

Interrupts are electrical signals multiplexed by the **interrupt controller**

- Sent on a specific pin of the CPU

Once an interrupt is received, a dedicated function is executed

- **Interrupt handler**

The kernel/user space can be interrupted at (nearly) any time to process an interrupt

Recap: Advanced PIC (APIC, I/O APIC)

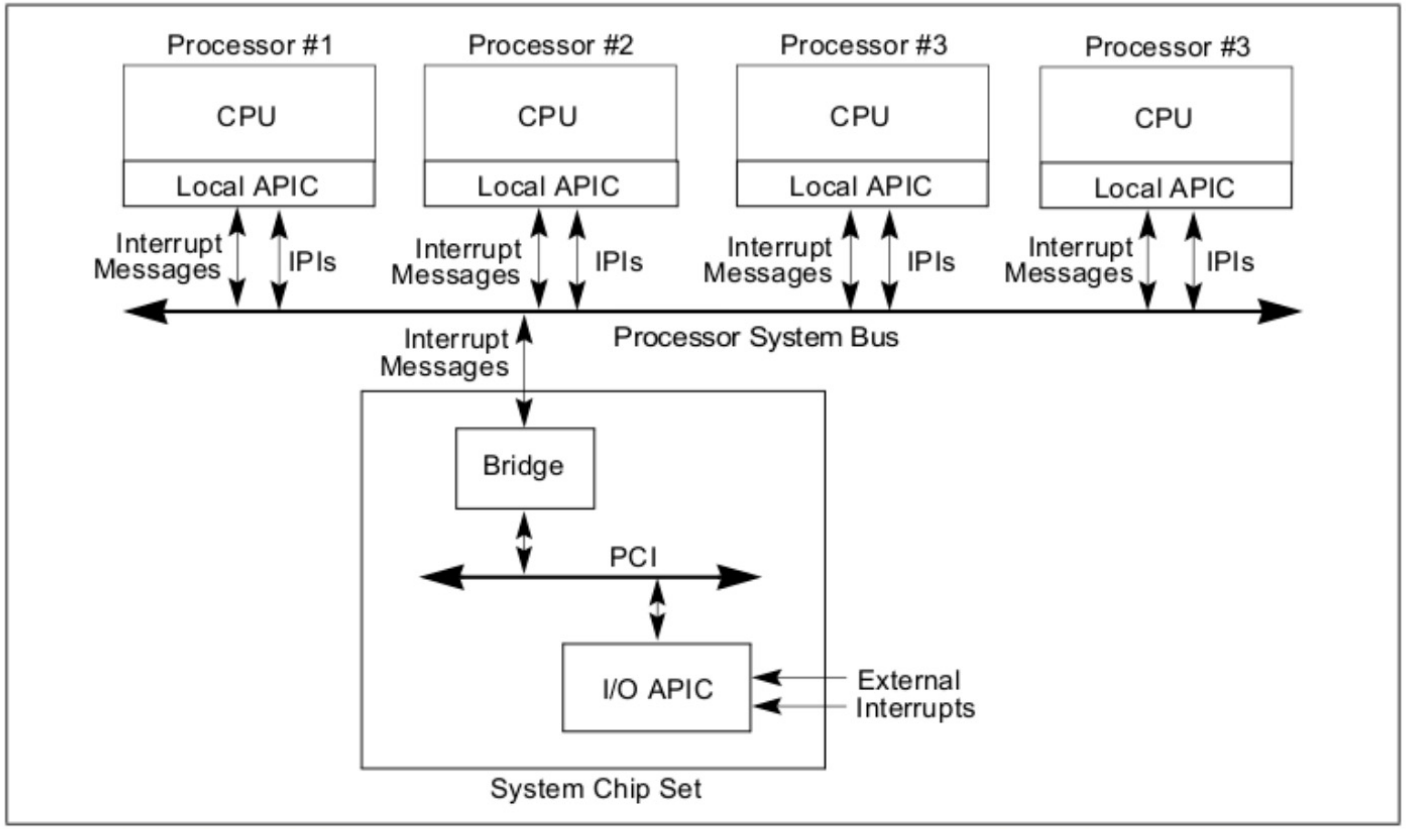
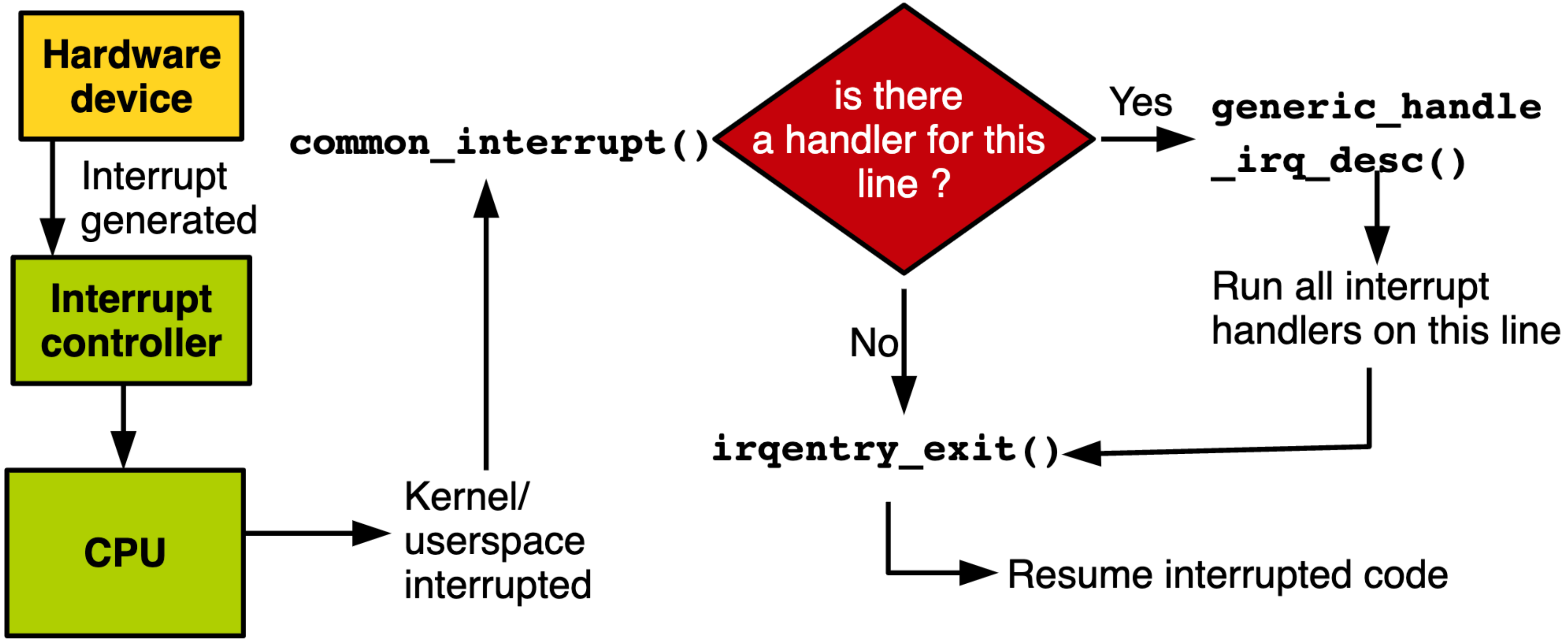


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

Recap: Interrupt handling in Linux



Today: interrupt handler

Top-halves (interrupt handlers) must run as quickly as possible

- They are interrupting other kernel/user code
- They are often timing-critical because they deal with hardware.
- They run in interrupt context: they cannot block
- One or all interrupt lines are disabled

Defer the less critical part of interrupt processing to a **bottom-half**

The history of bottom halves

"Top-half" and "bottom-half" are generic terms not specific to Linux

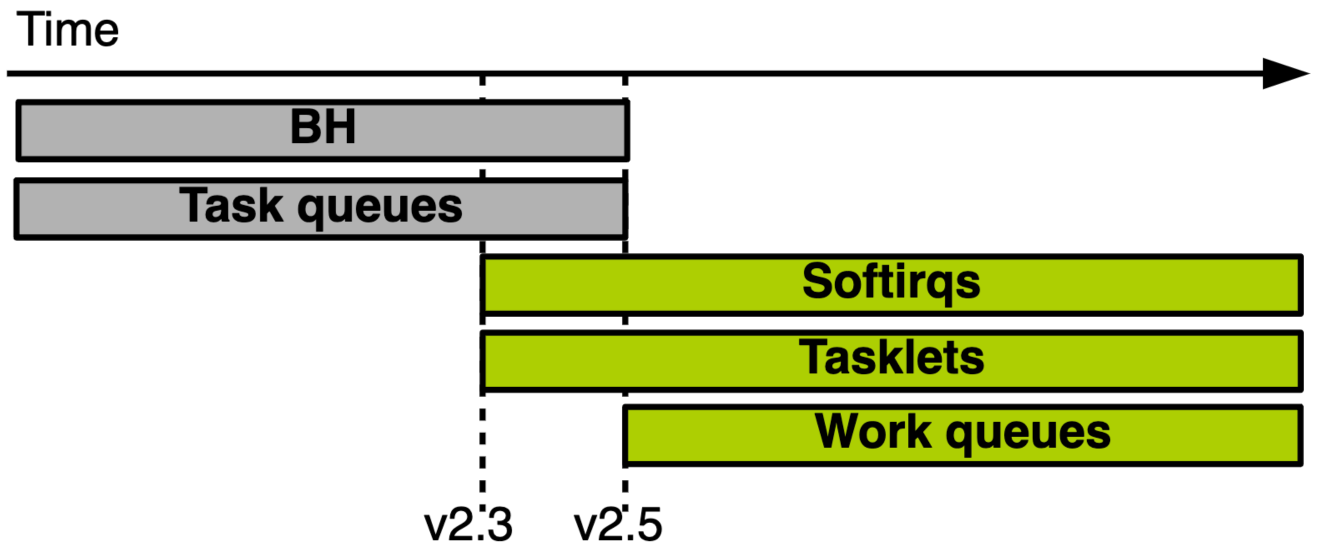
old "Bottom-Half" (BH) mechanism

- a statistically created list of 32 bottom halves

Task queues: queues of function pointers

- still too inflexible
- not lightweight enough for performance-critical subsystems (e.g., networking)

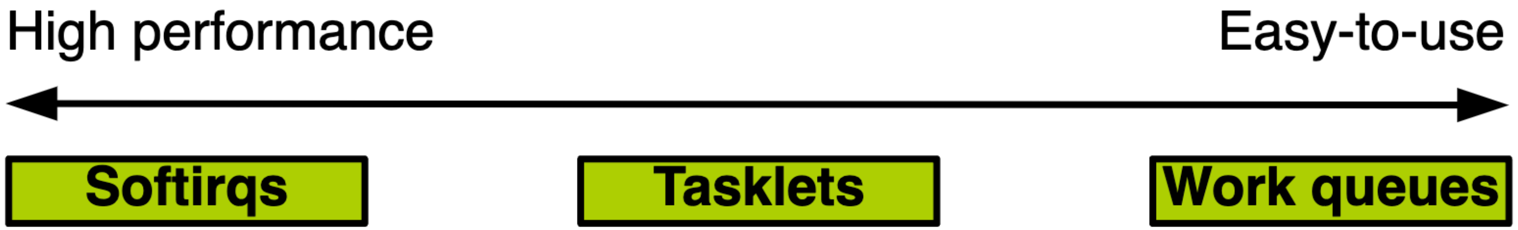
The history of bottom halves



BH → Softirq, tasklet

Task queue → work queue

Today’s bottom halves in Linux



All bottom-half mechanisms **run with all interrupts enabled**

Softirqs and **tasklets** run in an interrupt context

- Softirq is rarely used directly
- Tasklet is a simple and easy-to-use softirq (built on softirq)

Work queues run in a process context

- They can block and go to sleep

Softirq

```c
/* kernel/softirq.c */
/* Softirq is statically allocated at compile time */
static struct softirq_action softirq_vec[NR_SOFTIRQS]; /* softirq vector */
/* include/linux/interrupt.h */
enum {
    HI_SOFTIRQ=0,      /* [highest priority] high-priority tasklet */
    TIMER_SOFTIRQ,     /* timer */
    NET_TX_SOFTIRQ,    /* send network packets */
    NET_RX_SOFTIRQ,    /* receive network packets */
    BLOCK_SOFTIRQ,     /* block devices */
    IRQ_POLL_SOFTIRQ,  /* interrupt-poll handling for block device */
    TASKLET_SOFTIRQ,   /* normal priority tasklet */
    SCHED_SOFTIRQ,     /* scheduler */
    HRTIMER_SOFTIRQ,   /* unused */
    RCU_SOFTIRQ,       /* [lowest priority] RCU locking */

    NR_SOFTIRQS        /* the number of defined softirq (< 32) */
};
struct softirq_action {
    void   (*action)(struct softirq_action *);       /* softirq handler */
};
```

Execute softirq

Raise a softirq (by the h/w interrupt handler)

- Mark the execution of a particular softirq is needed
- Usually, a top-half marks its softirq for execution before returning

Pending softirqs are checked and executed in following places:

- In the return from hardware interrupt code path
- In the ksoftirqd kernel thread
- In any code that explicitly checks for and executes pending softirqs

Execute softirq

Go over the softirq vector and executes the pending softirq handler

```
/* kernel/softirq.c */
/* do_softirq() calls __do_softirq() */
void __do_softirq(void) /* much simplified version for explanation */
{
    u32 pending;
    pending = local_softirq_pending(); /* 32-bit flags for pending softirq */
    if (pending) {
        struct softirq_action *h;

        set_softirq_pending(0); /* reset the pending bitmask */
        h = softirq_vec;
        do {
            if (pending & 1)
                h->action(h); /* execute the handler of the pending softirq */
            h++;
            pending >>= 1;
        } while (pending);
    }
}
```

Use softirq: assign an index

```
enum {                          /* include/linux/interrupt.h */
  HI_SOFTIRQ=0,                 /* [highest priority] high-priority tasklet */
  TIMER_SOFTIRQ,                /* timer */
  NET_TX_SOFTIRQ,               /* send network packets */
  NET_RX_SOFTIRQ,               /* receive network packets */
  BLOCK_SOFTIRQ,                /* block devices */
  IRQ_POLL_SOFTIRQ,             /* interrupt-poll handling for block device */
  TASKLET_SOFTIRQ,              /* normal priority tasklet */
  SCHED_SOFTIRQ,                /* scheduler */
  HRTIMER_SOFTIRQ,              /* unused */
  RCU_SOFTIRQ,                  /* [lowest priority] RCU locking */
  YOUR_NEW_SOFTIRQ,             /* TODO: add your new softirq index, but not recommended */
  NR_SOFTIRQS                   /* the number of defined softirq (< 32) */
}
```

Use softirq: register a handler

```
/* kernel/softirq.c */
/* register a softirq handler for nr */
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}

/* net/core/dev.c */
static int __init net_dev_init(void)
{
    /* ... */
    /* register softirq handler to send messages */
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);

    /* register softirq handler to receive messages */
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
    /* ... */
}
```

handler for network message send

Use softirq: register a handler

Run with interrupts enabled and cannot sleep

The key advantage of softirq over tasklet is scalability

- If the same softirq is raised again while it is executing, another processor can run it simultaneously

This means that any shared data needs proper locking

- To avoid locking, most softirq handlers resort to per-processor data (data unique to each processor and thus not requiring locking)

Use softirq: raise a softirq

Softirqs are most often raised from within interrupt handlers (i.e., top halves)

- The interrupt handler performs the basic hardware-related work, raises the softirq, and then exits

```
/* include/linux/interrupt.h */
/* Disable interrupt and raise a softirq */
extern void raise_softirq(unsigned int nr);

/* Raise a softirq. Interrupt must already be off. */
extern void raise_softirq_irqoff(unsigned int nr);

/* kernel/time/timer.c */
run_local_timers() --> raise_softirq(TIMER_SOFTIRQ);
```

Tasklet

Built on top of softirqs

- `HI_SOFTIRQ`: high priority tasklet
- `TASKLET_SOFTIRQ`: normal priority tasklet

Running in an interrupt context (i.e., cannot sleep)

- Like softirq, **all interrupts** are **enabled**

Restricted concurrency than softirq

- The same tasklet cannot run concurrently

tasklet_struct

```c
/* include/linux/interrupt.h */
struct tasklet_struct {
   struct tasklet_struct *next;    /* next tasklet in the list */
   unsigned long state;            /* state of a tasklet: scheduled, running */
   atomic_t count;                 /* disable counter: != 0 cannot run   */
   void (*func)(unsigned long);/* tasklet handler function */
   unsigned long data;             /* argument of the tasklet function */
};
```

Schedule a tasklet

Scheduled tasklets are stored in two **per-processor linked list**:

- `tasklet_vec`, `tasklet_hi_vec`

```
/* kernel/softirq.c*/
struct tasklet_head {
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};

/* regular tasklet */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
/* high-priority tasklet */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

Schedule a tasklet

/ include/linux/interrupt.h, kernel/softirq.c */*

/ Schedule a regular tasklet; For high-priority tasklet, use tasklet_hi_schedule() */*

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
        __tasklet_schedule_common(t, &tasklet_vec,
                                  TASKLET_SOFTIRQ);
}
EXPORT_SYMBOL(__tasklet_schedule);
```

Schedule a tasklet

```
static void __tasklet_schedule_common(struct tasklet_struct *t,
                                      struct tasklet_head __percpu *headp,
                                      unsigned int softirq_nr)
{
        struct tasklet_head *head;
        unsigned long flags;

        local_irq_save(flags);               /* disable interrupt */
        head = this_cpu_ptr(headp);          /* Append this tasklet at the end of list */
        t->next = NULL;
        *head->tail = t;
        head->tail = &(t->next);
        raise_softirq_irqoff(softirq_nr); /* tasklet is a softirq */
        local_irq_restore(flags);            /* enable interrupt */
}
```

Tasklet softirq handlers

```
/* kernel/softirq.c*/
void __init softirq_init(void)
{
    /* ... */
    /* Tasklet softirq handlers are registered at initializing softirq */
    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}

static __latent_entropy void tasklet_action_comment(struct softirq_action *a,
                                                    struct tasklet_head *tl_head,
                                                    unsigned int softirq_nr)
{
    struct tasklet_struct *list;

    /* Clear the list for this processor by setting it equal to NULL */
    local_irq_disable();
    list = tl_head->head;
    tl_head->head = NULL;
    tl_head->tail = &tl_head->head;
    local_irq_enable();
```

Tasklet softirq handlers (cont'd)

```
/* For all tasklets in the list */
while (list) {
    struct tasklet_struct *t = list;
    list = list->next;
    /* If a tasklet is not processing and it is enabled */
    if (tasklet_trylock(t) && !atomic_read(&t->count)) {
            /* and it is not running */
            if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                BUG();
            /* then execute the associate tasklet handler */
            t->func(t->data);
            tasklet_unlock(t);
            continue;
        }
        tasklet_unlock(t);
    }
    local_irq_disable();
    t->next = NULL;
    *tl_head->tail = t;
    tl_head->tail = &t->next;
    __raise_softirq_irqoff(softirq_nr);
    local_irq_enable();
}
```

Use tasklet: declaring a tasklet

```
/* include/linux/interrupt.h */
/* Static declaration of a tasklet with initially enabled */
#define DECLARE_TASKLET(name, _callback) \
struct tasklet_struct name = { .count = ATOMIC_INIT(0), /* disable counter */\
                               .callback = _callback, \
                               .use_callback = true, }
/* Static declaration of a tasklet with initially disabled */
#define DECLARE_TASKLET_DISABLED(name, _callback) \
struct tasklet_struct name = { .count = ATOMIC_INIT(1), /* disable counter */\
                               .callback = _callback, \
                               .use_callback = true, }
/* Dynamic initialization of a tasklet */
extern void tasklet_init(struct tasklet_struct *t,
                         void (*func)(unsigned long), unsigned long data);
```

Use tasklet: tasklet handler

Run with interrupts enabled and cannot sleep

- If your tasklet shared data with an interrupt handler, you must task precautions (e.g., disable interrupt or obtain a lock)

Two of the same tasklets never run concurrently

- Because `tasklet_action()` checks `TASKLET_STATE_RUN`

But two different tasklets can run at the same time on two different processors

Schedule a tasklet

```
/* include/linux/interrupt.h */
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
/* Disable a tasklet by increasing the disable counter */
void tasklet_disable(struct tasklet_struct *t)
{
  tasklet_disable_nosync(t);
  tasklet_unlock_wait(t);             /* and wait until the tasklet finishes */
  smp_mb();
}
void tasklet_enable(struct tasklet_struct *t)
{
 smp_mb__before_atomic();
 atomic_dec(&t->count);
}
```

Tasklet example

Blackboard lec11-bottom-half.tar.gz

ksoftirqd

Per-processor kernel thread that aids processing softirqs (kernel softirq daemon)

If the number of softirqs grows excessive, the kernel wakes up `ksoftirqd` with normal priority (nice 0)

- No starvation of user-space application
- Running a softirq has the normal priority (nice 0)

```
▸ ps ax -eo pid,nice,stat,cmd | grep ksoftirq
     13    0 S     [ksoftirqd/0]
     22    0 S     [ksoftirqd/1]
     28    0 S     [ksoftirqd/2]
```

UIC COMPUTER SCIENCE

Work queues

Work queues defer work into a kernel thread

- Always runs in process context
- Thus, work queues are schedulable and can therefore sleep

By default, per-cpu kernel thread is created, kworker/n

- You can create additional per-CPU worker thread, if needed

Workqueues users can also create their own threads for better performance and lighten the load on default threads

```
▸ ps ax -eo pid,nice,stat,cmd | grep kworker
     8 -20 I<   [kworker/0:0H-events_highpri]
    17   0 I    [kworker/0:1-events]
    24 -20 I<   [kworker/1:0H-events_highpri]
```

Work queue: data structure

```c
/* kernel/workqueue.c */
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 */
    struct list_head worklist;  /* L: list of pending works */
    int nr_workers;             /* L: total number of workers */
    /* ... */
};
```

```c
/* include/workqueue.h */
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

UIC COMPUTER SCIENCE

Work queue: work thread

Worker threads execute the `worker_thread()` function

Infinite loop doing the following:

1. Check if there is some work to do in the current pool
2. If so, execute all the `work_struct` objects pending in the pool `worklist` by calling `process_scheduled_works()`
   - Call the `work_struct` function pointer `func`
   - `work_struct` objects removed
3. Go to sleep until a new work is inserted in the work queue

Work queue: create work

```
/* include/linux/workqueue.h */
/* Statically creating a work */
DECLARE_WORK(work, handler_func);
/* Dynamically creating a work at runtime */
INIT_WORK(work_ptr, handler_func);

/* Work handler prototype */
typedef void (*work_func_t)(struct work_struct *work);
/* Create/destory a new work queue in addition to the default queue  */
struct workqueue_struct *create_workqueue(char *name);
void destroy_workqueue(struct workqueue_struct *wq);
```

UIC COMPUTER SCIENCE

Work queue: schedule work

```
/* Put work task in global workqueue (kworker/n) */
bool schedule_work(struct work_struct *work);
bool schedule_work_on(int cpu,
                      struct work_struct *work);  /* on the specified CPU */

/* Queue work on a specified workqueue */
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);  /* on the specified CPU */
```

Work queue: finish work

```
/* Flush a specific work_struct */
int flush_work(struct work_struct *work);
/* Flush a specific workqueue: */
void flush_workqueue(struct workqueue_struct *);
/* Flush the default workqueue (kworkers): */
void flush_scheduled_work(void);

/* Cancel the work */
void flush_workqueue(struct workqueue_struct *wq);
/* Check if a work is pending */
work_pending(struct work_struct *work);
```

Work queue example

Blackboard lec11-bottom-half.tar.gz

Choose the right bottom-half

Bottom half	Context	Inherent serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Work queue	Process	None

All these generally run with interrupts enabled

If there is a shared data with an interrupt handler (top-half), need to disable interrupts or use locks

Disable softirq and tasklet

```
/* Disable softirq and tasklet processing on the local processor */
void local_bh_disable();
/* Eanble softirq and tasklet processing on the local processor */
void local_bh_enable();
```

These calls can be nested

- Only the final call to `local_bh_enable()` enables bottom halves

These calls do not disable workqueues processing

Next steps

hw6: CPU profiler (part 1)

- due Feb 16^{th}

Reading assignment:

- Optimizing Storage Performance with Calibrated Interrupts, OSDI'21
- due Feb 19^{th}

Further readings

LKD3: Chapter 8: Bottom Halves and Deferring Work