Linux Kernel Programming

Bottom-Halves and Deferring Work

Pierre Olivier

Systems Software Research Group @ Virginia Tech

March 2, 2017

Outline

Outline

Bottom-halves: general information

Presentation

- **Top-halves (interrupt handlers) must be as quick as possible**
  - Because they interrupt kernel/user code
    - Affects performance
  - Because they run with one/all lines disabled
    - Processing network traffic should not prevent the kernel from receiving keystrokes
- **Top-halves run in interrupt context: they cannot block**
  - Limit what they can do
- When processing an interrupt, ***the less-critical part of the work is deferred to a bottom-half***
  - Runs *later* (regarding the moment the actual interrupt occurs)

Bottom-halves: general information

Which part of the job in which half, reason of being

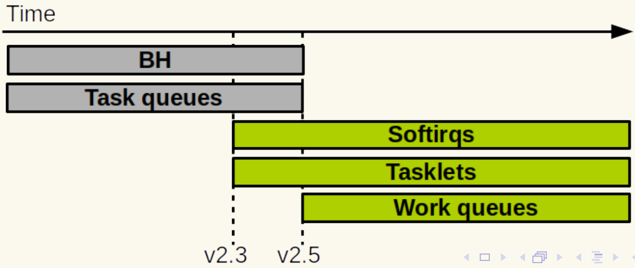
- **Work is time sensitive?** $\rightarrow$ top-half
- **Work is related to the hardware?** $\rightarrow$ top-half
- **Work should not be interrupted by another/the same interrupt?** $\rightarrow$ top-half
- **Everything else:** $\rightarrow$ **bottom-half**

- Bottom-halves run *later*
  - They generally run very soon after the actual interrupt
  - Crucial point is to run with interrupts enabled

Bottom-halves: general information

A bit of history

- Old *"**Bottom-Half**"* (BH) mechanism
  - 32 of them, globally synchronized: only one running in the system at the same time
    - Performance bottleneck
- ***Task queues***: queues of function pointers (handlers)
  - Handlers run at various time depending on which queue they are on
  - Not sufficient for performance critical subsystems (ex: networking)
- **BH and task queues were removed in 2.5**



Outline

Softirqs

Implementing softirqs, softirq execution

- **Softirqs**
  - Bottom-half with the best performance
  - `kernel/softirq.c`
  - Rarely used directly (tasklets instead)
    - However, tasklets are build upon softirqs

- `include/linux/interrupt.h`:

```
struct softirq_action {
  void (*action)(struct softirq_action *);
}
```

- `kernel/interrupt.c`:

```
static struct softirq_action
  softirq_vec[NR_SOFTIRQS];
/* NR_SOFTIRQ is max 32 */
```

- **Softirq handler**:

```
void handler_name(struct softirq_action *);
```

Softirqs

Implementing softirqs, softirq execution (2)

- The kernel runs a softirq by executing the handler:

```
/* let's assume my_softirq is a struct softirq_action * */
my_softirq->action(my_softirq);
```

- **Softirq execution**
  - Once registered, a softirq must be *raised* to indicate it needs to execute
    - Generally done by the top-half handler
  - **Raised softirqs are checked and executed:**
    - **In return from interrupt**
    - **In a kernel thread, `ksoftirqd`**
    - **In any code that explicitly checks for and runs raised softirqs** (`do_softirq()`)
- Check and run is done in `__do_softirq()`
  - Goes over the softirq array (`softirq_vec`)
  - Executes the handlers of raised (pending) softirqs

Softirqs

Using softirqs: softirqs indexes

- **Softirqs are declared statically at compile time**
  - `enum` in `linux/interrupt.h`

```
enum {
  HI_SOFTIRQ=0,      /* 0 */
  TIMER_SOFTIRQ,     /* 1 */
  NET_TX_SOFTIRQ,    /* 2 */
  NET_RX_SOFTIRQ,    /* 3 */
  BLOCK_SOFTIRQ,     /* 4 */
  IRQ_POLL_SOFTIRQ,  /* 5 */
  TASKLET_SOFTIRQ,   /* 6 */
  SCHED_SOFTIRQ,     /* 7 */
  HRTIMER_SOFTIRQ    /* 8 */
  RCU_SOFTIRQ,       /* 9 */
  NR_SOFTIRQ
};
```

  - Create an entry in this array to add a softirq to Linux
  - Entries ranked by priority (`HI_SOFTIRQ` is the highest)
    - **This is the order in which the array is iterated for softirq execution**

Softirqs

Using softirqs (2)

- (Very) simplified version of `__do_softirq()`:

```
int i;

/* Iterate in priority order */
for(i = 0; i < NR_SOFTIRQ; i++) {
  struct softirq_action *handler = softirq_vec[i];
  int pending = is_pending(handler);

  if(pending) {
    handler->(action(handler));
  }
}
```

Softirqs

Using softirqs: handler registration

- Handler registration done through `open_softirq()`:

```
open_softirq(SOFTIRQ_INDEX, softirq_handler);
/* real example (net/core/dev.c): */
open_softirq(NET_TX_SOFTIRQ, net_tx_action);
```

- **Softirqs run in interrupt context**
  - Cannot block/sleep
- When a softirq handler is running, **softirqs are disabled on the local core**
  - However softirqs can run concurrently on different cores
    - Including softirqs with the same index → **shared data must be protected against concurrent accesses** (generally use per-processor data)
  - **Good scalability vs tasklets**
    - If the same softirq is raised while its handler is running it can executes on another core
    - One tasklet cannot run concurrently on multiple cores

Softirqs

Using softirqs: raising a softirq

- `raise_softirq()`:

```
/* kernel/time/timer.c: */
raise_softirq(TIMER_SOFTIRQ);
```

  - Disables interrupts on the local core before marking the softirq as pending
    - With `local_irq_save()`
  - Restores them to the previous state afterward
    - With `local_irq_restore()`
  - Generally called from the interrupt handler (top-half)
  - Optimization if interrupts are already off:

```
/* net/core/dev.c: */
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

Softirqs

Softirq example

- Cannot be registered from module (softirq symbols not exported)
  - Adding a softirq must be done from inside the kernel code
- `include/linux/interrupt.h`:

```
diff -rc linux-4.10.1/include/linux/interrupt.h linux-4.10.1.modified/include/linux/interrupt.
    h
*** linux-4.10.1/include/linux/interrupt.h  2017-02-26 10:09:33.000000000 +0000
--- linux-4.10.1.modified/include/linux/interrupt.h 2017-02-28 15:57:46.088406158 +0000
***************
*** 456,461 ****
--- 456,462 ----
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ, /* Unused, but kept as tools rely on the
            numbering. Sigh! */
+   PIERRE_SOFTIRQ,
    RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

    NR_SOFTIRQS
```

Softirqs

Softirq example (2)

- `kernel/main.c`:

```
diff -rc linux-4.10.1/init/main.c linux-4.10.1.modified/init/main.c
*** linux-4.10.1/init/main.c  2017-02-26 10:09:33.000000000 +0000
--- linux-4.10.1.modified/init/main.c  2017-02-28 16:18:02.672173235 +0000
***************
*** 89,94 ****
--- 89,100 ----
  #include <asm/sections.h>
  #include <asm/cacheflush.h>

+
+ void pierre_softirq_handler(struct softirq_action * action)
+ {
+         printk("Pierre softirq running!\n");
+ }
+
  static int kernel_init(void *);

  extern void init_IRQ(void);
```

```
***************
*** 669,674 ****
--- 675,686 ----

    ftrace_init();

+
+
+   open_softirq(PIERRE_SOFTIRQ, pierre_softirq_handler);
+   printk("Raising Pierre softirq\n");
+   raise_softirq(PIERRE_SOFTIRQ);
+
    /* Do the rest non-__init'ed, we're now alive */
    rest_init();
  }
```

Outline

Tasklets

Implementing tasklets

- **Tasklets** are implemented on top of softirqs
  - Same behavior, simpler interface, less locking rules
    - **The same tasklet cannot run concurrently on multiple cores**
  - Implemented in `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` softirqs
- A tasklet is represented by a `tasklet_struct` (`include/linux/interrupt.h`):

```
struct tasklet_struct
{
  struct tasklet_struct *next;
  unsigned long state;
  atomic_t count;
  void (*func)(unsigned long);
  unsigned long data;
}
```

- `func`: handler (args: `data`)
- `next`: linked list of tasklets
- `state`: `enum` (scheduled to run/running)
- `count`:
  - `!=0`: disabled, cannot run
  - `0`: enabled, can be marked pending

Tasklets

Implementing tasklets: scheduling tasklets

- *Scheduling* tasklets == *raising* softirqs
- Scheduled tasklets put in two per-processor linked lists:
  - `tasklet_hi_vec` (high priority)
  - `tasklet_vec`
- Scheduling a tasklet is done through `tasklet_schedule(t)` or `tasklet_hi_schedule(t)`:
  1. Check if the tasklet is already scheduled, if not, call `__tasklet_schedule(t)`
  2. Disable interrupts
  3. Add the tasklet to the corresponding linked list
  4. Raise `TASKLET_SOFTIRQ` or `HI_SOFTIRQ`
  5. Restore interrupts and return

Tasklets

Implementing tasklets: tasklets softirqs handlers

Tasklet softirqs handlers:

- `tasklet_action()` and `tasklet_hi_action()`
  1. Clear the list for the current CPU and iterate over its content, for each raised tasklet:
     1. Check `TASKLET_STATE_RUN` flag
     2. If the tasklet is not running, set that flag
     3. Check the `count` value (is it enabled?)
     4. Run the tasklet and clear the `TASKLET_STATE_RUN` flag

Tasklets

Using tasklets: creation

- **Creation:**

```
/* Statically: */
DECLARE_TASKLET(tasklet_name, handler_name, handler_arguments);
DECLARE_TASKLET_DISABLED(tasklet_name, handler_name, handler_arguments);

/* Dynamically */
tasklet_ptr = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
tasklet_init(tasklet_ptr, handler_name, handler_arguments);
```

- Difference between `DECLARE_TASKLET()` and `DECLARE_TASKLET_DISABLED()`:
  - `count` field of the initialized `struct task_struct`
    - 1 for *enabled*, 0 for *disabled* (will not run even if it is scheduled to)

Tasklets

Using tasklets: handler

- **Handler**
  - Prototype:

```
void handler_name(unsigned long data);
```

- Like softirqs, **tasklets cannot sleep** (run in interrupt context)
  - No use of blocking semaphores, no kmalloc with `GFP_KERNEL`, etc.
- **Tasklets run with interrupts enabled**
  - Shared data with an interrupt handler? *disable interrupts/get a (non-sleeping) lock*
  - Two different tasklets can run concurrently on different cores: *need locking if a tasklet shares data with another tasklet or a softirq*

Tasklets

Using tasklets: scheduling the tasklet to run, enabling/disabling a tasklet

- To mark the tasklet as pending: `tasklet_schedule()`

```
tasklet_schedule(&tasklet_name); /* tasklet_name being of type struct tasklet */
```

  - Runs one in a near future on the same CPU it is schedule from
    - Independently of the number of calls to `tasklet_schedule()`

- **Disabling/enabling a tasklet:**

```
tasklet_disable(&tasklet_name);
tasklet_enable(&tasklet_name);
```

  - Set the `count` member of the `struct tasklet`
  - `tasklet_disable()` blocks until any potential running handler finishes
    - `tasklet_disable_nosync()` in order not to wait, probably unsafe
    - `tasklet_enable()` must be called after declaring a tasklet through `DELCARE_TASKLET_DISABLED()`

Tasklets

ksoftirqd

- System can be flooded by softirqs (and tasklets)
  - Scenarios with high interrupts arrival frequency
  - Plus softirqs can reactivate themselves
- **Takes a lot of CPU time from userspace processes**
  - **But softirqs still need to be processed promptly**
- A solution is to defer *reactivated* softirqs until the next time softirq run
  - Generally at the next interrupt occurrence
  - Sub-optimal on an idle system
- **Solution:** defer reactivated softirqs into per-cpu, low priority kernel threads: `ksoftirqd`
  - No starvation of CPU time for user space processes in case of highly frequent interrupts
    - `ksoftirqd` priority is nice 19
  - On an idle system, `ksoftirqd` is scheduled quickly

Tasklets

Tasklet example

- Tasklet example in a module:

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/interrupt.h>

void my_tasklet_handler(unsigned long data
      );

static DECLARE_TASKLET(my_tasklet,
      my_tasklet_handler, 0);

void my_tasklet_handler(unsigned long data
      )
{
  printk("tasklet executing.\n");
}
```

```c
static int __init my_mod_init(void)
{
  printk("Entering module.\n");
  printk("Scheduling tasklet.\n");

  tasklet_schedule(&my_tasklet);

  return 0;
}

static void __exit my_mod_exit(void)
{
  tasklet_disable(&my_tasklet);
  printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
```

Outline

Workqueues

Presentation

- **Workqueues run in *process context***
  - They are a schedulable entity, so **workqueues can sleep**
    - Needed when the bottom-half need to allocate a lot of memory, sleep on a lock (semaphore/mutex), perform blocking I/O
    - In these situations softirqs and tasklets cannot be used → **use workqueues**
- Work deferred into work queues is handled by **kernel threads**:
  - Default, per-cpu ones: `kworker/n` (before: `events/n`) where `n` is the CPU id
    - Syntax: `kworker/<cpu>[unbound]:<id><priority>`
  - Workqueues users can also create their own threads
    - Better performance & lighten the load on default threads

Workqueues

Work queues implementation: `worker_pool` and `work_struct`

- `kernel/workqueue.c`:

```c
struct worker_pool {
  spinlock_t       lock;
  int              cpu;
  int              node;
  int              id;
  unsigned int     flags;
  /* list of work_struct: */
  struct list_head worklist;
  /* number of associated worker threads: */
  int              nr_workers;


  /* ... */
};
```

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

- `include/linux/workqueue.h`:

```c
typedef void (*work_func_t)(struct work_struct *work);
```

Workqueues

Work queues implementation: worker thread function

- Worker threads execute the `worker_thread()` function (`kernel/workqueue.c`)
- 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`, passing a pointer the the `work_struct` itself as a parameter
     - `work_struct` objects removed from the list after being processed
  3. Go to sleep
     - Worker threads are awaken next time some work is inserted in the workqueue

Workqueues

Using work queues

- Creating / destroying a **workqueue**:

```
struct workqueue_struct *create_workqueue(char *name);
void destroy_workqueue(struct workqueue_struct *);
```

- Creating a **work entity**:

```
/* statically: */
DECLARE_WORK(work, func);
DECLARE_DELAYED_WORK(work, func);
DECLARE_DEFERRABLE_WORK(work, func);
```

```
/* dynamically */
INIT_WORK{work_ptr, func);
INIT_DELAYED_WORK(work_ptr, func);
INIT_DEFERRABLE_WORK(work_ptr, func);
```

  - `work` is the name of the initialized `work_struct`
  - `work_ptr` is a pointer to an allocated `work_struct`
  - `func` is the name of the handler function
  - `DELAYED` is related to work item which execution can be delayed by a given time after they are scheduled to run
  - `DEFERRED` indicates low priority work items

- Handler prototype:

```
void handler(struct work_struct *work);
```

Workqueues

Using work queues (2)

- Creating a work entity, example:

```
static void my_handler(struct work_struct *work)
{
  /* ... */
}

/* Static creation: */
DECLARE_WORK(work_item, my_handler);

static int __init my_init_function(void)
{
  /* dynamic creation: */
  struct work_struct * work_item2;

  work_item2 = kmalloc(sizeof(struct work_item), GFP_KERNEL);
  INIT_WORK(work_item2, my_handler);
}
```

Workqueues

Using work queues (3)

- Enqueueing a work on a specific, previously created work queue:

```
int queue_work (struct workqueue_struct *wq, struct work_struct *work);
int queue_work_on (int cpu, struct workqueue_struct *wq, struct work_struct *work);
int queue_delayed_work (struct workqueue_struct *wq, struct delayed_work *work),
     unsigned long delay);
int queue_delayed_work_on (int cpu, struct workqueue_struct *wq, struct delayed_work *
     work), unsigned long delay);
```

- Enqueue on the default kernel threads (`kworker`s):

```
int schedule_work (struct work_struct *):
int schedule_work_on (int cpu, struct work_struct *):
int scheduled_delayed_work (struct delayed_work *, unsigned long delay);
int scheduled_delayed_work_on (int cpu, struct delayed_work *, unsigned long delay);
```

- **Work executes next time a worker thread is awaken**
- Can schedule on a specific core ("`_on`" functions)
- Can delay execution by a given number of timer ticks (`delay`)

Workqueues

Using work queues: work function parameters

- **How to pass parameters to the work item handler?**
  - Not done directly like in tasklets, *but the* `work_struct` *is passed as a parameter to its own handler execution*
  - Solution: **put the `work_struct` in a data structure**
    - Parameter as another member of the data structure
    - Use `container_of()` from inside the handler

```
struct work_item {
  struct work_struct ws;
  int parameter;
};

static void handler( struct work_struct *work)
{
  struct work_item *wi = (struct work_item *)container_of(work, struct work_item, ws);
  int parameter = wi->parameter;
}
```

- In that case (`work_struct` as the first field of the containing data structure), one can also do:

```
struct work_item *wi = (struct work_item *)work;
```

Workqueues

Using work queues: utility functions

- To ensure work finishes its execution

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

- Canceling scheduled work:

```
int cancel_work_sync(struct work_struct *work);
int cancel_delayed_work_sync(struct delayed_work *dwork);
```

- Checking if work is pending:

```
work_pending(work);        /* work is struct work_struct */
delayed_work_pending(work);   /* same thing */
```

- Destroying a workqueue:

```
void destroy_workqueue(struct workqueue_struct *);
```

Workqueues

Workqueue module example

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/workqueue.h>

struct work_item {
  struct work_struct ws;
  int parameter;
};

struct work_item *wi, *wi2;
struct workqueue_struct *my_wq;

static void handler( struct work_struct *
      work)
{
  int parameter = ((struct work_item *)
      container_of(work, struct work_item,
       ws))->parameter;
  printk("doing some work ...\n");
  printk("parameter is: %d\n", parameter);
}
```

```c
static int __init my_mod_init(void)
{
  printk("Entering module.\n");

  my_wq = create_workqueue("pierre_wq");
  wi = kmalloc(sizeof(struct work_item),
      GFP_KERNEL);
  wi2 = kmalloc(sizeof(struct work_item),
      GFP_KERNEL);

  INIT_WORK(&wi->ws, handler);
  wi->parameter = 42;
  INIT_WORK(&wi2->ws, handler);
  wi2->parameter = -42;

  schedule_work(&wi->ws);
  queue_work(my_wq, &wi2->ws);

  return 0;
}
```

Workqueues

Workqueue module example (2)

```
static void __exit my_mod_exit(void)
{
  flush_scheduled_work();
  flush_workqueue(my_wq);
  kfree(wi);
  kfree(wi2);
  destroy_workqueue(my_wq);
  printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
MODULE_LICENSE("GPL");
```

Outline

Using the right bottom-half and misc. information

Using the right bottom-half

- Work has a lot of potential for parallelism and/or is very performance critical
  - **Softirqs**
    - Can run concurrently on multiple cores (need to take care of concurrency issues)
    - High performance: generally run very quickly after marked pending
- No need for parallelism but performance still important
  - **Tasklets**
    - Two tasklets cannot run concurrently on several cores
    - Performance close to softirqs (depends on the tasklet type considered)
- Need to run in process context
  - **Workqueues**
    - Can sleep
    - Less performance, need to wait to be scheduled to perform work

Using the right bottom-half and misc. information

Locking between bottom-halves

- **Softirqs**
  - Need intra-softirq (same softirq) locking (thread-safe)
  - Need inter-softirq (different softirqs) locking in case of shared data between them
- **Tasklets**
  - No need to protect against concurrent accesses from the same tasklet
  - Two different tasklets sharing data need proper locking (inter-tasklets locking)
- **Workqueues**
  - Intra- and inter-workqueue locking needed
- **All of these generally run with interrupts enabled**
  - In case of shared data with an interrupt handler (top-half), disabling interrupts + locking needed

Using the right bottom-half and misc. information

Disabling bottom-half processing

- To disable / enable softirqs and tasklets on the local core:

```
void local_bh_disable();
void local_bh_enable();
```

  - Can nest
    - Need to call `local_bh_enable()` as much times as `local_bh_disabled()` was called to re-enable interrupts
- These calls do not disable workqueues processing
  - Not a problem as these run in process context and do not happen asynchronously like softirqs/tasklets

Outline

Additional sources of information

Disabling bottom-half processing

- Numerous details about the internals of Linux interrupt management (top/bottom half): `https://0xax.gitbooks.io/linux-insides/content/interrupts/` (Linux 3.18)
- Workqueues documentation in Linux sources: `Documentation/workqueues.txt`