developerWorks®

Kernel APIs, Part 2: Deferrable functions, kernel tasklets, and work queues

An introduction to bottom halves in Linux 2.6

M. Tim Jones
Independent author

02 March 2010

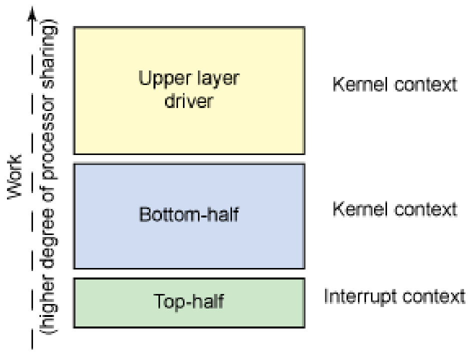
For high-frequency threaded operations, the Linux® kernel provides tasklets and work queues. Tasklets and work queues implement deferrable functionality and replace the older bottom-half mechanism for drivers. This article explores the use of tasklets and work queues in the kernel and shows you how to build deferrable functions with these APIs.

View more content in this series

This article explores a couple of methods used to defer processing between kernel contexts (specifically, within the 2.6.27.14 Linux kernel). Although these methods are specific to the Linux kernel, the ideas behind them are useful from an architectural perspective, as well. For example, you could implement these ideas in traditional embedded systems in place of a traditional scheduler for work scheduling.

Before diving into the methods used in the kernel to defer functions, however, let's start with some background on the problem being solved. When an operating system is interrupted because of a hardware event (such as the presence of a packet through a network adapter), the processing begins in an interrupt. Typically, the interrupt kicks off a substantial amount of work. Some amount of this work is done in the context of the interrupt, and work is passed up the software stack for additional processing (see Figure 1).

Figure 1. Top-half and bottom-half processing



The question is, how much work should be done in the interrupt context? The problem with interrupt context is that some or all interrupts can be disabled during this time, which increases the latency of handling other hardware events (and introduces changes in processing behavior). Therefore, minimizing the work done in the interrupt is desirable, pushing some amount of the work into the kernel context (where there is a higher likelihood that the processor can be gainfully shared).

As shown in Figure 1, the processing done in the interrupt context is called the *top half,* and interrupt-based processing that's pushed outside of the interrupt context is called the *bottom half* (where the top half schedules the subsequent processing by the bottom half). The bottom-half processing is performed in the kernel context, which means that interrupts are enabled. This leads to better performance because of the ability to deal quickly with high-frequency interrupt events by deferring non-time-sensitive work.

Short history of bottom halves

Linux kernel version

This discussion of tasklets and work queues uses the 2.6.27.14 version of the Linux kernel.

Linux tends to be a Swiss Army knife of functionality, and deferring functionality is no different. Since kernel 2.3, softirqs have been available that implement a set of 32 statically defined bottom halves. As static elements, these are defined at compile time (unlike the new mechanisms, which are dynamic). Softirqs were used for time-critical processing (software interrupts) in the kernel thread context. You can find the source to the softirq functionality in ./kernel/softirq.c. Also introduced in the 2.3 Linux kernel are tasklets (see ./include/linux/interrupt.h). Tasklets are built on top of softirqs to allow dynamic creation of deferrable functions. Finally, in the 2.5 Linux kernel, work queues were introduced (see ./include/linux/workqueue.h). Work queues permit work to be deferred outside of the interrupt context into the kernel process context.

Let's now explore the dynamic mechanisms for work deferral, tasklets, and work queues.

Introducing tasklets

Softirqs were originally designed as a vector of 32 softirq entries supporting a variety of software interrupt behaviors. Today, only nine vectors are used for softirqs, one being the `TASKLET_SOFTIRQ` (see ./include/linux/interrupt.h). And although softirqs still exist in the kernel, tasklets and work queues are recommended instead of allocating new softirq vectors.

Tasklets are a deferral scheme that you can schedule for a registered function to run later. The top half (the interrupt handler) performs a small amount of work, and then schedules the tasklet to execute later at the bottom half.

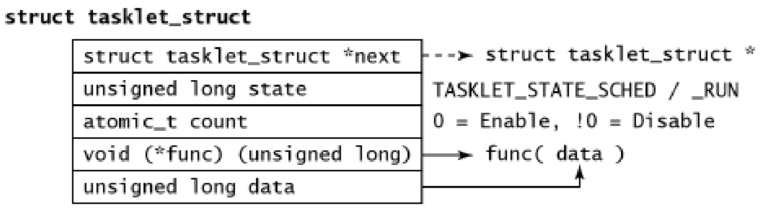
Listing 1. Declaring and scheduling a tasklet

```
/* Declare a Tasklet (the Bottom-Half) */
void tasklet_function( unsigned long data );

DECLARE_TASKLET( tasklet_example, tasklet_function, tasklet_data );

...

/* Schedule the Bottom-Half */
tasklet_schedule( &tasklet_example );
```

A given tasklet will run on only one CPU (the CPU on which the tasklet was scheduled), and the same tasklet will never run on more than one CPU of a given processor simultaneously. But different tasklets can run on different CPUs at the same time.

Tasklets are represented by the tasklet_struct structure (see Figure 2), which includes the necessary data to manage and maintain the tasklet (state, enable/disable via an `atomic_t`, function pointer, data, and linked-list reference).

Figure 2. The internals of the tasklet_struct structure



Tasklets are scheduled through the softirq mechanism, sometimes through ksoftirqd (a per-CPU kernel thread), when the machine is under heavy soft-interrupt load. The next section explores the various functions available in the tasklets application programming interface (API).

Embedded systems heritage

The ideas behind tasklets and work queues have some heritage in embedded systems. In many embedded systems, there is no traditional scheduler, only work deferral (driven by input/ouput [I/O] or internal processing). In lieu of a scheduler, interrupts and applications defer work as a means of scheduling processing later by other elements of the system. In this way, the scheduler becomes a processor of work queues (feeding work to handler functions) or bit masks (which indicate the ability of a tasklet to do its work).

Tasklets API

Tasklets are defined using a macro called `DECLARE_TASKLET` (see Listing 2). Underneath, this macro simply provides a `tasklet_struct` initialization of the information you provide (tasklet name, function, and tasklet-specific data). By default, the tasklet is enabled, which means that it can be scheduled. A tasklet can also be declared as disabled by default using the `DECLARE_TASKLET_DISABLED` macro. This requires that the `tasklet_enable` function be invoked to make the tasklet schedulable. You can enable and disable a tasklet (from a scheduling perspective) using the `tasklet_enable` and `tasklet_disable` functions, respectively. A `tasklet_init` function also exists that initializes a `tasklet_struct` with the user-provided tasklet data.

Listing 2. Tasklet creation and enable/disable functions

```
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 * );
```

Two disable functions exist, each of which requests a disable of the tasklet, but only the `tasklet_disable` returns after the tasklet has been terminated (where the `tasklet_disable_nosync` may return before the termination has occurred). The disable functions allow the tasklet to be "masked" (that is, not executed) until the enable function is called. Two enable functions also exist: one for normal priority scheduling (`tasklet_enable`) and one for enabling higher-priority scheduling (`tasklet_hi_enable`). The normal-priority schedule is performed through the `TASKLET_SOFTIRQ`-level softirq, where high priority is through the `HI_SOFTIRQ`-level softirq.

As with the normal and high-priority enable functions, there are normal and high-priority schedule functions (see Listing 3). Each function enqueues the tasklet on the particular softirq vector (`tasklet_vec` for normal priority and `tasklet_hi_vec` for high priority). Tasklets from the high-priority vector are serviced first, followed by those on the normal vector. Note that each CPU maintains its own normal and high-priority softirq vectors.

Listing 3. Tasklet scheduling functions

```
void tasklet_schedule( struct tasklet_struct * );
void tasklet_hi_schedule( struct tasklet_struct * );
```

Finally, after a tasklet has been created, it's possible to stop a tasklet through the `tasklet_kill` functions (see Listing 4). The `tasklet_kill` function ensures that the tasklet will not run again and, if the tasklet is currently scheduled to run, will wait for its completion, and then kill it. The `tasklet_kill_immediate` is used only when a given CPU is in the dead state.

Listing 4. Tasklet kill functions

```
void tasklet_kill( struct tasklet_struct * );
void tasklet_kill_immediate( struct tasklet_struct *, unsigned int cpu );
```

From the API, you can see that the tasklet API is simple, and so is the implementation. You can find the implementation of the tasklet mechanism in ./kernel/softirq.c and ./include/linux/interrupt.h.

Simple tasklet example

Let's look at a simple usage of the tasklets API (see Listing 5). As shown here, a tasklet function is created with associated data (`my_tasklet_function` and `my_tasklet_data`), which is then used to declare a new tasklet using `DECLARE_TASKLET`. When the module is inserted, the tasklet is scheduled, which makes it executable at some point in the future. When the module is unloaded, the `tasklet_kill` function is called to ensure that the tasklet is not in a schedulable state.

Listing 5. Simple example of a tasklet in the context of a kernel module

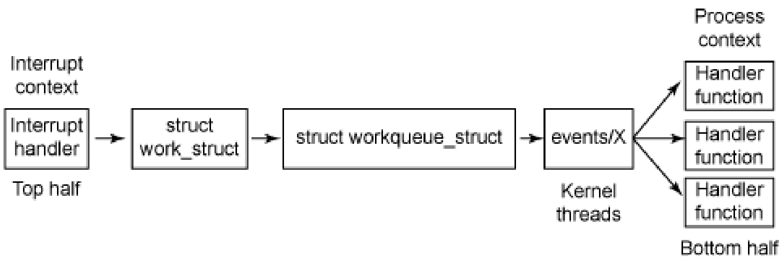
```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/interrupt.h>

MODULE_LICENSE("GPL");

char my_tasklet_data[]="my_tasklet_function was called";

/* Bottom Half Function */
void my_tasklet_function( unsigned long data )
{
  printk( "%s\n", (char *)data );
  return;
}

DECLARE_TASKLET( my_tasklet, my_tasklet_function,
   (unsigned long) &my_tasklet_data );

int init_module( void )
{
  /* Schedule the Bottom Half */
  tasklet_schedule( &my_tasklet );

  return 0;
}

void cleanup_module( void )
{
  /* Stop the tasklet before we exit */
  tasklet_kill( &my_tasklet );

  return;
}
```

Introducing work queues

Work queues are a more recent deferral mechanism, added in the 2.5 Linux kernel version. Rather than providing a one-shot deferral scheme as is the case with tasklets, work queues are a generic deferral mechanism in which the handler function for the work queue can sleep (not possible in the tasklet model). Work queues can have higher latency than tasklets but include a richer API for work deferral. Deferral used to be managed by task queues through `keventd` but is now managed by kernel worker threads named `events/X`.

Work queues provide a generic method to defer functionality to bottom halves. At the core is the work queue (struct `workqueue_struct`), which is the structure onto which work is placed. Work is represented by a `work_struct` structure, which identifies the work to be deferred and the deferral function to use (see Figure 3). The `events/X` kernel threads (one per CPU) extract work from the work queue and activates one of the bottom-half handlers (as indicated by the handler function in the struct `work_struct`).

Figure 3. The process behind work queues



As the `work_struct` indicates the handler function to use, you can use the work queue to queue work for a variety of handlers. Now, let's look at the API functions that can be found for work queues.

Work queue API

The work queue API is slightly more complicated that tasklets, primarily because a number of options are supported. Let's first explore the work queues, and then we'll look at work and the variants.

Recall from Figure 3 that the core structure for the work queue is the queue itself. This structure is used to enqueue work from the top half to be deferred for execution later by the bottom half. Work queues are created through a macro called `create_workqueue`, which returns a `workqueue_struct` reference. You can remote this work queue later (if needed) through a call to the `destroy_workqueue` function:

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

The work to be communicated through the work queue is defined by the `work_struct` structure. Typically, this structure is the first element of a user's structure of work definition (you'll see an example of this later). The work queue API provides three functions to initialize work (from an allocated buffer); see Listing 6. The `INIT_WORK` macro provides for the necessary initialization and setting of the handler function (passed by the user). In cases where the developer needs a delay before the work is enqueued on the work queue, you can use the `INIT_DELAYED_WORK` and `INIT_DELAYED_WORK_DEFERRABLE` macros.

Listing 6. Work initialization macros

```
INIT_WORK( work, func );
INIT_DELAYED_WORK( work, func );
INIT_DELAYED_WORK_DEFERRABLE( work, func );
```

With the work structure initialized, the next step is enqueuing the work on a work queue. You can do this in a few ways (see Listing 7). First, simply enqueue the work on a work queue using `queue_work` (which ties the work to the current CPU). Or, you can specify the CPU on which the handler should run using `queue_work_on`. Two additional functions provide the same functionality for delayed work (whose structure encapsulates the `work_struct` structure and a timer for work delay).

Listing 7. Work queue functions

```
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 *dwork, unsigned long delay );

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

You can use a global kernel-global work queue, with four functions that address this work queue. These functions (shown in Listing 8) mimic those from Listing 7, except that you don't need to define the work queue structure.

Listing 8. Kernel-global work queue functions

```
int schedule_work( struct work_struct *work );
int schedule_work_on( int cpu, struct work_struct *work );

int scheduled_delayed_work( struct delayed_work *dwork, unsigned long delay );
int scheduled_delayed_work_on(
  int cpu, struct delayed_work *dwork, unsigned long delay );
```

There are also a number of helper functions that you can use to flush or cancel work on work queues. To flush a particular work item and block until the work is complete, you can make a call to `flush_work`. All work on a given work queue can be completed using a call to `flush_workqueue`. In both cases, the caller blocks until the operation is complete. To flush the kernel-global work queue, call `flush_scheduled_work`.

```
int flush_work( struct work_struct *work );
int flush_workqueue( struct workqueue_struct *wq );
void flush_scheduled_work( void );
```

You can cancel work if it is not already executing in a handler. A call to `cancel_work_sync` will terminate the work in the queue or block until the callback has finished (if the work is already in progress in the handler). If the work is delayed, you can use a call to `cancel_delayed_work_sync`.

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

Finally, you can find out whether a work item is pending (not yet executed by the handler) with a call to `work_pending` or `delayed_work_pending`.

```
work_pending( work );
delayed_work_pending( work );
```

That's the core of the work queue API. You can find the implementation of the work queue API in ./kernel/workqueue.c, with API definitions in ./include/linux/workqueue.h. Let's now continue with a simple example of the work queue API.

Simple work queue example

The following example illustrates a few of the core work queue API functions. As with the tasklets example, you implement this example in the context of a kernel module for simplicity.

First, look at your work structure and the handler function that you'll use to implement the bottom half (see Listing 9). The first thing you'll note here is a definition of your work queue structure reference (`my_wq`) and the `my_work_t` definition. The `my_work_t` typedef includes the `work_struct` structure at the head and an integer that represents your work item. Your handler (a callback function) de-references the `work_struct` pointer back to the `my_work_t` type. After emitting the work item (integer from the structure), the work pointer is freed.

Listing 9. Work structure and bottom-half handler

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

MODULE_LICENSE("GPL");

static struct workqueue_struct *my_wq;

typedef struct {
  struct work_struct my_work;
  int    x;
} my_work_t;

my_work_t *work, *work2;


static void my_wq_function( struct work_struct *work)
{
  my_work_t *my_work = (my_work_t *)work;

  printk( "my_work.x %d\n", my_work->x );

  kfree( (void *)work );

  return;
}
```

Listing 10 is your `init_module` function, which begins with creation of the work queue using the `create_workqueue` API function. Upon successful creation of the work queue, you create two work items (allocated via `kmalloc`). Each work item is then initialized with `INIT_WORK`, the work defined, and then enqueued onto the work queue with a call to `queue_work`. The top-half process (simulated here) is now complete. The work will then, at some time later, be processed by the handler as shown in Listing 10.

Listing 10. Work queue and work creation

```
int init_module( void )
{
  int ret;

  my_wq = create_workqueue("my_queue");
  if (my_wq) {
    /* Queue some work (item 1) */
    work = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);
    if (work) {

      INIT_WORK( (struct work_struct *)work, my_wq_function );

      work->x = 1;

      ret = queue_work( my_wq, (struct work_struct *)work );

    }

    /* Queue some additional work (item 2) */
    work2 = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);
    if (work2) {

      INIT_WORK( (struct work_struct *)work2, my_wq_function );

      work2->x = 2;

      ret = queue_work( my_wq, (struct work_struct *)work2 );

    }

  }

  return 0;
}
```

The final elements are shown in Listing 11. Here, in module cleanup, you flush the particular work queue (which blocks until the handler has completed processing of the work), and then destroy the work queue.

Listing 11. Work queue flush and destruction

```
void cleanup_module( void )
{
  flush_workqueue( my_wq );

  destroy_workqueue( my_wq );

  return;
}
```

Differences between tasklets and work queues

From this short introduction to tasklets and work queues, you can see two different schemes for deferring work from top halves to bottom halves. Tasklets provide a low-latency mechanism that is simple and straightforward, while work queues provide a flexible API that permits queuing of multiple work items. Each defers work from the interrupt context, but only tasklets run atomically in a run-to-complete fashion, where work queues permit handlers to sleep, if necessary. Either method is useful for work deferral, so the method selected is based on your particular needs.

Going further

The work-deferral methods explored here represent the historical and current methods used in the Linux kernel (excluding timers, which will be covered in a future article). They are certainly not new—in fact, they have existed in other forms in the past—but they represent an interesting architectural pattern that is useful in Linux and elsewhere. From softirqs to tasklets to work queues to delayed work queues, Linux continues to evolve in all areas of the kernel while providing a consistent and compatible user space experience.

Resources

Learn

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Much of the information available on tasklets and work queues on the Internet tends to be a bit dated. For an introduction to the rework of the work queue API, check out this nice introduction from LWN.net. You can also learn some useful information from the Linux Device Drivers book.
- *I'll do it later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers* (PDF) by Matthew Wilcox is a great introduction to the various deferral mechanisms in Linux.
- This useful presentation from Jennifer Hou (of the Computer Science department at the University of Illinois at Urbana-Champaign) provides a nice overview of the Linux kernel, an introduction to softirqs, and a short introduction to work deferral with tasklets.
- This synopsis from the Advanced Systems Programming course at the University of San Francisco (taught by Professor Emeritus, Dr. Allan Cruse) provides a great set of resources for work deferral (including the concepts that drive them).
- The seminal Linux Device Drivers text provides a useful introduction to work deferral (PDF). In this free chapter ("Timers, Delays, and Deferred Work"), you'll find a deep (though slightly dated) discussion of tasklets and work queues.
- For more information on where and when to use tasklets versus work queues, check out this exchange on the kernel mail list.
- Columbia University instructor Junfeng Yang provides a great introduction to interrupts and system calls in Linux (PDF) in this presentation.
- This presentation from Dr. Tai-Yi Huang of the National Tsing-Hua University of Taiwan provides a great introduction to interrupts and exceptions with the Linux kernel (PowerPoint). In addition, this presentation explores the topics of softirqs, tasklets, and work queues, including example code.
- Browse all of Tim's articles on developerWorks.
- In the developerWorks Linux zone, find hundreds of articles, tutorials, discussion forums, and a wealth other resources for Linux developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools as well as IT industry trends.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Follow developerWorks on Twitter.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.