KERNEL MECHANISMS

Important Notice

This courseware is both the product of the author and of freely available opensource materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of this courseware cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - source code and binaries (where applicable) - that form part of this courseware, and that are present on the participant CD, are released under the GNU GPL v2 license and can therefore be used subject to terms of the afore-mentioned license. If you do use any of them, in any manner, you will also be required to clearly attribute their original source (author of this courseware and/or other copyright/trademark holders).

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant CD are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Table of Contents

<< Source / Refer: Text provided to participants - "Linux Kernel Development" by Robert Love, 2nd Ed., Novell Press, Ch. 10 "Timers and Time Management". >>

The notions of kernel time, HZ, jiffies, etc have been covered earlier – to relook at this, peek into Ch. 10.

Timers

Timers - sometimes called dynamic timers or kernel timers - are essential for managing the flow of time in kernel code. Kernel code often needs to delay execution of some function until a later time. In previous chapters, we looked at using the bottom-half mechanisms, which are great for deferring work until later. Unfortunately, the definition of later is intentionally quite vague. The purpose of bottom halves is not so much to delay work, but simply to not do the work now. What we need is a tool for delaying work a specified amount of time - certainly no less, and with hope, not much longer. The solution is kernel timers.

A timer is very easy to use. You perform some initial setup, specify an expiration time, specify a function to execute upon said expiration, and activate the timer. The given function will run after the timer expires. Timers are not cyclic. The timer is destroyed after it expires. This is one reason for the dynamic nomenclature[7]: Timers are constantly created and destroyed, and there is no limit on the number of timers. Timers are very popular throughout the entire kernel.

[7] Another reason is because of the existence of static timers in older (pre-2.3) kernels. They were created at compile-time, not runtime. They were lame, so no one shed tears over their demise.

Using Timers

Timers are represented by struct timer_list, which is defined in <*linux/timer.h*>:

```
struct timer_list {
 struct list_head entry;  /* entry in linked list of timers */
 unsigned long expires;   /* expiration value, in jiffies */
 spinlock_t lock;         /* lock protecting this timer */
 void (*function)(unsigned long);/* the timer handler function */
 unsigned long data;      /* lone argument to the handler */
 struct tvec_t_base_s *base;  /* internal timer field, do not touch */
};
```

Fortunately, the usage of timers requires little understanding of this data structure. In fact, toying with it is discouraged to keep code forward compatible with changes. The kernel

provides a family of timer-related interfaces to make timer management easy. Everything is declared in <*linux/timer.h*>. Most of the actual implementation is in kernel/timer.c. The first step in creating a timer is defining it:

```
struct timer_list my_timer;
```

Next, the timer's internal values must be initialized. This is done via a helper function and must be done prior to calling any timer management functions on the timer:

```
init_timer(&my_timer);
```

Now you fill out the remaining values as required:

```
my_timer.expires = jiffies + delay;    /* timer expires in delay ticks */
my_timer.data = 0;/* zero is passed to the timer handler */
my_timer.function = my_function;       /* function to run when timer expires
*/
```

The my_timer.expires value specifies the timeout value in absolute ticks. When the current jiffies count is equal to or greater than my_timer.expires, the handler function my_timer.function is run with the lone argument of my_timer.data. As you can see from the timer_list definition, the function must match this prototype:

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

The data parameter enables you to register multiple timers with the same handler, and differentiate between them via the argument. If you do not need the argument, you can simply pass zero (or any other value). Finally, you activate the timer:

```
add_timer(&my_timer);
```

And, voila, the timer is off and running! Note the significance of the expired value. The kernel runs the timer handler when the current tick count is equal to or greater than the specified expiration. Although the kernel guarantees to run no timer handler prior to the timer's expiration, there may be a delay in running the timer. Typically, timers are run fairly close to their expiration; however, they might be delayed until the first timer tick after their expiration. Consequently, timers cannot be used to implement any sort of hard real-time processing.

<< Actually, it could be worse than that: it's even possible to “drop” an interval when using add_timer(). See this post on StackOverflow for a lot more detail.
Ultimately, the solution to precision timing on the Linux kernel? HRT – High Resolution Timers.

For user-space (to use HRT):
“ … The hrtimer patch converts the following kernel functionality to use hrtimers:

- nanosleep
- itimers
- posix-timers”

>>

Sometimes you might need to modify the expiration of an already active timer. The kernel implements a function, mod_timer(), which changes the expiration of a given timer:

```
mod_timer(&my_timer, jiffies + new_delay); /* new
                                              expiration */
```

The mod_timer() function can operate on timers that are initialized but not active, too. If the timer is inactive, mod_timer() activates it. The function returns zero if the timer was inactive and one if the timer was active. In either case, upon return from mod_timer(), the timer is activated and set to the new expiration.If you need to deactivate a timer prior to its expiration, use the del_timer() function:

```
del_timer(&my_timer);
```

The function works on both active and inactive timers. If the timer is already inactive, the function returns zero; otherwise, the function returns one. Note that you do not need to call this for timers that have expired because they are automatically deactivated. A potential race condition that must be guarded against exists when deleting timers. When del_timer() returns, it guarantees only that the timer is no longer active (that is, that it will not be executed in the future). On a multiprocessing machine, however, the timer handler might already be executing on another processor. To deactivate the timer and wait until a potentially executing handler for the timer exits, use del_timer_sync():

```
del_timer_sync(&my_timer);
```

Unlike del_timer(), del_timer_sync() cannot be used from interrupt context.

Timer Race Conditions

Because timers run asynchronously with respect to the currently executing code, several potential race conditions exist. First, never do the following as a substitute for a mere mod_timer(), because this is unsafe on multiprocessing machines:

```
del_timer(my_timer) ;
my_timer->expires = jiffies + new_delay;
add_timer(my_timer);
```

Second, in almost all cases, you should use del_timer_sync() over del_timer(). Otherwise, you cannot assume the timer is not currently running, and that is why you made the call in the first place! Imagine if, after deleting the timer, the code went on to free or otherwise manipulate resources used by the timer handler. Therefore, the synchronous version is preferred.

Finally, you must make sure to protect any shared data used in the timer handler function. The kernel runs the function asynchronously with respect to other code. Data with a timer should be protected as discussed in Chapters 8 and 9.

The Timer Implementation

The kernel executes timers in bottom-half context, as softirqs, after the timer interrupt completes. The timer interrupt handler runs update_process_times(), which calls run_local_timers():

```
void run_local_timers(void)
{
        raise_softirq(TIMER_SOFTIRQ);
}
```

The TIMER_SOFTIRQ softirq is handled by run_timer_softirq(). This function runs all the expired timers (if any) on the current processor.

Timers are stored in a linked list. However, it would be unwieldy for the kernel to either constantly traverse the entire list looking for expired timers, or keep the list sorted by expiration value; the insertion and deletion of timers would then become very expensive.

Instead, the kernel partitions timers into five groups based on their expiration value. Timers move down through the groups as their expiration time draws closer. The partitioning ensures that, in most executions of the timer softirq, the kernel has to do little work to find the expired timers. Consequently, the timer management code is very efficient.

Delaying Execution

Often, kernel code (especially drivers) needs a way to delay execution for some time without using timers or a bottom-half mechanism. This is usually to allow hardware time to complete a given task. The time is typically quite short. For example, the specifications for a network card might list the time to change Ethernet modes as two microseconds. After setting the desired speed, the driver should wait at least the two microseconds before continuing.

The kernel provides a number of solutions, depending on the semantics of the delay. The solutions have different characteristics. Some hog the processor while delaying effectively preventing the accomplishment of any real work. Other solutions do not hog the processor, but offer no guarantee that your code will resume in exactly the required time[8].

[8] Actually, no approach guarantees that the delay will be for exactly the time requested. Some come extremely close, however - and they all promise to wait at least as long as needed. Some just wait longer.

Busy Looping

The simplest solution to implement (although rarely the optimal solution) is busy waiting or busy looping. This technique works only when the time you want to delay is some integer multiple of the tick rate or precision is not very important.The idea is simple: Spin in a loop until the desired number of clock ticks pass. For example,

```
unsigned long delay = jiffies + 10;        /* ten ticks */
 while (time_before(jiffies, delay))
        ;
```

<< The time_before() function returns true as long as parameter 1 is less than parameter 2 (in terms of time); once parameter 1 is >= parameter 2 it returns false. >>

The loop continues until jiffies is larger than delay, which will occur only after 10 clock ticks have passed. On x86 with HZ equal to 1000, this results in a wait of 10 milliseconds. Similarly,

```
unsigned long delay = jiffies + 2*HZ;        /* two seconds */
 while (time_before(jiffies, delay))
        ;
```

This will spin until 2*HZ clock ticks has passed, which is always two seconds regardless

of the clock rate. This approach is not nice to the rest of the system. While your code waits, the processor is tied up spinning in a silly loop - no useful work is accomplished! In fact, you rarely want to take this brain-dead approach, and it is shown here because it is a clear and simple method for delaying execution. You might also encounter it in someone else's not-so-pretty code.

A better solution would be to reschedule your process to allow the processor to accomplish other work while your code waits:

```
unsigned long delay = jiffies + 5*HZ;

while (time_before(jiffies, delay))
        cond_resched();
```

The call to cond_resched() schedules a new process, but only if need_resched is set. In other words, this solution conditionally invokes the scheduler only if there is some more important task to run. Note that because this approach invokes the scheduler, you cannot make use of it from an interrupt handler - only from process context. In fact, all these approaches are best used from process context, because interrupt handlers should execute as quickly as possible (and busy looping does not help accomplish that goal!). Furthermore, delaying execution in any manner, if at all possible, should not occur while a lock is held or interrupts are disabled.

C aficionados might wonder what guarantee is given that the previous loops even work. The C compiler is usually free to perform a given load only once. Normally, no assurance is given that the jiffies variable in the loop's conditional statement is even reloaded on each loop iteration. The kernel requires, however, that jiffies be reread on each iteration, as the value is incremented elsewhere: in the timer interrupt. Indeed, this is why the variable is marked volatile in <*linux/jiffies.h*>. The volatile keyword instructs the compiler to reload the variable on each access from main memory and never alias the variable's value in a register, guaranteeing that the previous loop completes as expected.

Small Delays

Sometimes, kernel code (again, usually drivers) requires very short (smaller than a clock tick) and rather precise delays. This is often to synchronize with hardware, which again usually lists some minimum time for an activity to complete - often less than a millisecond. It would be impossible to use jiffies-based delays, as in the previous examples, for such a short wait. With a timer interrupt of 100Hz, the clock tick is a rather large 10 milliseconds! Even with a 1000Hz timer interrupt, the clock tick is still one millisecond.

Another solution is clearly necessary for smaller, more precise delays. Thankfully, the kernel provides two functions for microsecond and millisecond delays, both defined in `<linux/delay.h>`, which do not use jiffies:

```
void udelay(unsigned long usecs)
void mdelay(unsigned long msecs)
```

The former function delays execution by busy looping for the specified number of microseconds. The latter function delays execution for the specified number of milliseconds. Recall one second equals 1000 milliseconds, which equals 1,000,000 microseconds. Usage is trivial:

```
udelay(150);        /* delay for 150 µs */
```

The udelay() function is implemented as a loop that knows how many iterations can be executed in a given period of time. The mdelay() function is then implemented in terms of udelay(). Because the kernel knows how many loops the processor can complete in a second (see the sidebar on BogoMips), the udelay() function simply scales that value to the correct number of loop iterations for the given delay.

My BogoMIPS Are Bigger than Yours!

The BogoMIPS value has always been a source of confusion and humor. In reality, the BogoMIPS calculation has very little to do with the performance of your computer and is primarily used only for the udelay() and mdelay() functions. Its name is a contraction of bogus (that is, fake) and MIPS (million of instructions per second). Everyone is familiar with a boot message similar to the following (this is on a 1GHz Pentium 3):

```
Detected 1004.932 MHz processor.
Calibrating delay loop... 1990.65 BogoMIPS
```

The BogoMIPS value is the number of busy loop iterations the processor can perform in a given period. In effect, BogoMIPS are a measurement of how fast a processor can do nothing! This value is stored in the loops_per_jiffy variable and is readable from /proc/cpuinfo. The delay loop functions use the loops_per_jiffy value to figure out (fairly precisely) how many busy loop iterations they need to execute to provide the requisite delay.The kernel computes loops_per_jiffy on boot via calibrate_delay() in init/main.c.

The udelay() function should be called only for small delays because larger delays on fast machines might result in overflow. As a rule, do to not use udelay() for delays over one millisecond in duration. For longer durations, mdelay() works fine.

Like the other busy waiting solutions for delaying execution, neither of these functions

(especially mdelay(), because it is used for such long delays) should be used unless absolutely needed. Remember that it is rude to busy loop with locks held or interrupts disabled because system response and performance will be adversely affected. If you require precise delays, however, these calls are your best bet.

Typical uses of these busy waiting functions delay for a very small amount of time, usually in the microsecond range.

<<

Delay in nanoseconds:

```
void ndelay(unsigned long x);
```

!Note! CAREFUL: The ndelay() is rather coarse; see this Q&A on SO: "Why udelay and ndelay is not accurate in linux kernel?"

>>

schedule_timeout()

A more optimal method of delaying execution is to use schedule_timeout(). This call puts your task to sleep until at least the specified time has elapsed. There is no guarantee that the sleep duration will be exactly the specified time - only that the duration is at least as long as specified. When the specified time has elapsed, the kernel wakes the task up and places it back on the runqueue. Usage is easy:

```
/* set task's state to interruptible sleep */
set_current_state(TASK_INTERRUPTIBLE);

/* take a nap and wake up in "s" seconds */
schedule_timeout(s * HZ);
```

The lone parameter is the desired relative timeout, in jiffies. This example puts the task in interruptible sleep for 's' seconds. Because the task is marked TASK_INTERRUPTIBLE, it wakes up prematurely if it receives a signal. If the code does not want to process signals, you can use TASK_UNINTERRUPTIBLE instead. *The task must be in one of these two states before schedule_timeout() is called or else the task will not go to sleep.*

Note that because schedule_timeout() invokes the scheduler, code that calls it must be capable of sleeping. See Chapters 8 and 9 for discussions on atomicity and sleeping. In short, you must be in process context and must not hold a lock.

[FYI / OPTIONAL]

The schedule_timeout() function is fairly straightforward. Indeed, it is a simple application of kernel timers, so let's take a look at it:

```
signed long schedule_timeout(signed long timeout)
{
        timer_t timer;
        unsigned long expire;

        switch (timeout)
        {
        case MAX_SCHEDULE_TIMEOUT:
                schedule();
                goto out;
        default:
                if (timeout < 0)
                {
                        printk(KERN_ERR "schedule_timeout: wrong timeout "
                            "value %lx from %p\n", timeout,
                            __builtin_return_address(0));
                        current->state = TASK_RUNNING;
                        goto out;
                }
        }

        expire = timeout + jiffies;

        init_timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;

        add_timer(&timer);
        schedule();
        del_timer_sync(&timer);

        timeout = expire - jiffies;

 out:
        return timeout < 0 ? 0 : timeout;
}
```

The function creates a timer with the original name timer and sets it to expire in timeout clock ticks in the future. It sets the timer to execute the process_timeout() function when the timer expires. It then enables the timer and calls schedule(). Because the task is supposedly marked TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE, the scheduler does not run the task, but instead picks a new one.

When the timer expires, it runs process_timeout():

```
void process_timeout(unsigned long data)
{
        wake_up_process((task_t *) data);
}
```

This function puts the task in the TASK_RUNNING state and places it back on the runqueue.

When the task reschedules, it returns to where it left off in schedule_timeout() (right after the call to schedule()). In case the task was awakened prematurely (if a signal was received), the timer is destroyed. The function then returns the time slept.

The code in the switch() statement is for special cases and is not part of the general usage of the function. The MAX_SCHEDULE_TIMEOUT check enables a task to sleep indefinitely. In that case, no timer is set (because there is no bound on the sleep duration) and the scheduler is immediately invoked. If you do this, you had better have another method of waking your task up!

Sleeping on a Wait Queue, with a Timeout

Chapter 4 looked at how process context code in the kernel can place itself on a wait queue to wait for a specific event, and then invoke the scheduler to select a new task. Elsewhere, when the event finally occurs, wake_up() is called and the tasks sleeping on the wait queue are awakened and can continue running.

Sometimes it is desirable to wait for a specific event or wait for a specified time to elapse - whichever comes first. In those cases, code might simply call schedule_timeout() instead of schedule() after placing itself on a wait queue. The task wakes up when the desired event occurs or the specified time elapses. The code needs to check why it woke up - it might be because of the event occurring, the time elapsing, or a received signal - and continue as appropriate.

<< In fact, one could just use the timeout variant of a wait queue, and ensure that 'condition' never comes true:

```
wait_event_timeout(wait_queue_head_t queue,
                   int condition, long timeout)
wait_event_interruptible_timeout(wait_queue_head_t queue,
                  int condition, long timeout)
```

>>

<< Source / Refer: Text provided to participants - "Linux Kernel Development" by Robert Love, 2nd Ed., Novell Press, Ch. 7 "Bottom Halves and Deferring Work". >>

Some of the notes that follow below are extracted from:

Essential Linux Device Drivers

"Essential Linux Device Drivers"
Hardcover: 744 pages

- Author: Sreekrishnan Venkateswaran
- Publisher: Prentice Hall PTR Open Source Software Development Series; 1 edition (April 6, 2008)
- Language: English
- ISBN-10: 0132396556
- ISBN-13: 978-0132396554

Kernel Threads

A kernel thread is a way to implement background tasks inside the kernel. The task can be busy handling asynchronous events or sleep-waiting for an event to occur. Kernel threads are similar to user processes, except that they live in kernel space and have access to kernel functions and data structures. Like user processes, kernel threads have the illusion of monopolizing the processor because of preemptive scheduling.

Many device drivers utilize the services of kernel threads to implement assistant or helper tasks. For example, the khubd kernel thread, which is part of the Linux USB driver core

(covered in Chapter 11, "Universal Serial Bus") monitors USB hubs and configures USB devices when they are hot-plugged into the system.

SIDEBAR - Seeing Kernel Threads

'ps' can show us the running kernel threads on a Linux system.

```
$ cat /proc/version
Linux version 2.6.24-16-generic (buildd@palmer) (gcc version 4.2.3 (Ubuntu
4.2.3-2ubuntu7)) #1 SMP Thu Apr 10 13:23:42 UTC 2008
$ cat /etc/issue
Ubuntu 8.04.1 \n \l

$
```

The entries in the right-most column showing up in square brackets are kernel threads.

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:01 /sbin/init
    2 ?        S<     0:00 [kthreadd]
    3 ?        S<     0:00 [migration/0]
    4 ?        S<     0:00 [ksoftirqd/0]
    5 ?        S<     0:00 [watchdog/0]
    9 ?        S<     0:00 [events/0]
   11 ?        S<     0:00 [khelper]
   47 ?        S<     0:00 [kblockd/0]
   51 ?        S<     0:00 [kacpid]
   52 ?        S<     0:00 [kacpi_notify]
  144 ?        S<     0:00 [kseriod]
  186 ?        S      0:00 [pdflush]
  187 ?        S      0:00 [pdflush]
  188 ?        S<     0:00 [kswapd0]
  229 ?        S<     0:00 [aio/0]
 1537 ?        S<     0:00 [ksuspend_usbd]
 1541 ?        S<     0:00 [khubd]
 1562 ?        S<     0:00 [ata/0]
 1567 ?        S<     0:00 [ata_aux]
 2353 ?        S<     0:00 [scsi_eh_0]
 2354 ?        S<     0:00 [scsi_eh_1]
 2355 ?        S<     0:00 [scsi_eh_2]
 2356 ?        S<     0:00 [scsi_eh_3]
 2399 ?        S<     0:00 [scsi_eh_4]
 2400 ?        S<     0:00 [scsi_eh_5]
 2627 ?        S<     0:00 [kjournald]
 2855 ?        S<s    0:00 /sbin/udevd --daemon
 3227 ?        S<     0:00 [iwl3945/0]
 3297 ?        S<     0:00 [kpsmoused]
 4026 ?        S<     0:00 [irda_sir_wq]
 4159 ?        S<     0:00 [pccardd]
```

```
 4480 ?        S<     0:00 [kjournald]
 4483 ?        Ss     0:01 /sbin/mount.ntfs <...>
 4484 ?        S<     0:00 [kjournald]
 4485 ?        S<     0:00 [kjournald]
 4486 ?        S<     0:00 [kjournald]

--snip--

16111 ?        Ss     0:00 /usr/bin/dbus-daemon --fork <...>
16266 pts/9    Ss     0:00 bash
19740 ?        S<     0:00 [migration/1]
19741 ?        S<     0:00 [ksoftirqd/1]
19742 ?        S<     0:00 [watchdog/1]
19743 ?        S<     0:00 [kondemand/1]
19744 ?        S<     0:00 [iwl3945/1]
19745 ?        S<     0:00 [ata/1]
19746 ?        S<     0:00 [aio/1]
19747 ?        S<     0:00 [kblockd/1]
19748 ?        S<     0:00 [events/1]
22357 pts/9    R+     0:00 ps ax
$
```

The [ksoftirqd/0] kernel thread is an aid to implement softirqs. Softirqs are raised by interrupt handlers to request "bottom half" processing of portions of the handler whose execution can be deferred. We take a detailed look at bottom halves and softirqs in Chapter 4, "Laying the Groundwork," but the basic idea here is to allow as little code as possible to be present inside interrupt handlers. Having small interrupt handlers reduces interrupt-off times in the system, resulting in lower latencies. Ksoftirqd's job is to ensure that a high load of softirqs neither starves the softirqs nor overwhelms the system. On Symmetric Multi Processing (SMP) machines where multiple thread instances can run on different processors in parallel, one instance of ksoftirqd is created per CPU to improve throughput (ksoftirqd/n, where n is the CPU number).

The events/n threads (where n is the CPU number) help implement work queues, which are another way of deferring work in the kernel. Parts of the kernel that desire deferred execution of work can either create their own work queue or make use of the default events/n worker thread. Work queues are also dissected in Chapter 4.

The task of the pdflush kernel thread is to flush out dirty pages from the page cache. The page cache buffers accesses to the disk. To improve performance, actual writes to the disk are delayed until the pdflush daemon writes out dirtied data to disk. This is done if the available free memory dips below a threshold, or if the page has remained dirty for a sufficiently long time. In 2.4 kernels, these two tasks were respectively performed by separate kernel threads, bdflush and kupdated. You might have noticed two instances of pdflush in the ps output. A new instance is created if the kernel senses that existing instances have their hands full, servicing disk queues. This improves throughput, especially if your system has multiple disks and many of them are busy.

kjournald is the generic kernel journaling thread, which is used by filesystems such as EXT3.

The Linux Network File System (NFS) server is implemented using a set of kernel threads named nfsd.

Creating a Kernel Thread

...When you are comfortable with kernel threads, you can use them as a test vehicle for carrying out various experiments within the kernel.

Assume that you would like the kernel to asynchronously invoke a user mode program to send you an email or pager alert, whenever it senses that the health of certain key kernel data structures is deteriorating. (For instance, free space in network receive buffers has dipped below a low watermark.)

This is a candidate for being implemented as a kernel thread for the following reasons:

- It's a background task because it has to wait for asynchronous events.
- It needs access to kernel data structures because the actual detection of events is done by other parts of the kernel.
- It has to invoke a user mode helper program, which is a time-consuming operation.

To create a kernel thread, use kernel_thread() *:

```
ret = kernel_thread(mykthread, NULL,
            CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
```

** Several kernel APIs exist to create a kernel thread; this is one of them.*

The flags specify the resources to be shared between the parent and child threads. CLONE_FILES specifies that open files are to be shared, and CLONE_SIGHAND requests that signal handlers be shared.

The thread starts by invoking *daemonize()*, which performs initial housekeeping and changes the parent of the calling thread to a kernel thread called kthreadd. Each Linux thread has a single parent. If a parent process dies without waiting for its child to exit, the child becomes a zombie process and wastes resources. Reparenting the child to kthreadd, avoids this and ensures proper cleanup when the thread exits.[1]

[1] In 2.6.21 and earlier kernels, daemonize() reparents the calling thread to the init task by calling reparent_to_init().

Because daemonize() blocks all signals by default, use *allow_signal()* to enable delivery

if your thread desires to handle a particular signal. There are no signal handlers inside the kernel, so use *signal_pending()* to check for signals and take appropriate action. kernel_thread() is depreciated in favor of the higher-level kthread API, which is built over the former. We will look at kthreads later on.

Internals Viewpoint

<< An interesting point regards the memory descriptor (struct mm_struct * : part of the task_struct) of kernel threads. Since kernel threads run only in Kernel Mode, they never access virtual addresses below PAGE_OFFSET (0xc0000000).

This is achieved by having two fields in the process descriptor refer to the memory descriptor - *mm* and *active_mm*. Kernel threads do not have a "user-space" mapping -thus their *mm* field is always NULL; however, to refer their kernel-space Page Tables and kernel virtual addresses, they make use of the *active_mm* field.

So, a sure way to distinguish between user and kernel threads is to look up their *mm* value : if NULL => it's a kernel thread!
Also, if you find several tasks having the same “mm” field, they are threads of the same application.

Also, unlike user processes, kernel threads do not use memory regions (VMAs).
>>

Example

```
# cat kt1.c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include "convenient.h"

#define MODULE_NAME     "kt"

/* Our simple kernel thread. */
static int mykt(void *arg)
{
        /*
         * Daemonize: become a 'true' kernel thread: no attached user
         * resources,
         * close user-space pages (mm), block & flush all signals, 'inherit'
         * init's open files..etc.
```

```
         */
        daemonize("mykt");
        MSG("name: %s PID: %d TGID: %d\n",
                current->comm, current->pid, current->tgid);
        if (!current->mm)
                MSG("mm field NULL.\n");

        allow_signal(SIGINT);
        allow_signal(SIGQUIT);

        while (1) {
                MSG("K Thread %d going to sleep now...\n", current->pid);
                set_current_state (TASK_INTERRUPTIBLE);
                schedule();     // and yield the processor...
                // we're back on! has to be due to a signal
                if (signal_pending (current))
                        break;
        }
        // We've been interrupted by a signal...
        set_current_state (TASK_RUNNING);
        MSG("K thread %d exiting now...\n", current->pid);
        return 0;
}

static int hello_init(void)
{
        int ret;

        MSG("Create a kernel thread...\n");
        ret = kernel_thread( mykt, NULL, CLONE_FS | CLONE_FILES |
                 CLONE_SIGHAND | SIGCHLD);
           /* 2nd arg is (void * arg) to pass, ret val is PID on success */
        if (ret < 0) {
                printk(KERN_ERR "kt1: kernel_thread failed.");
                return ret;
        }

        MSG("Module %s initialized, thread PID %d alive.\n", MODULE_NAME, ret);
        return 0;
}

static void hello_exit(void)
{
        MSG("Module %s unloaded.\n", MODULE_NAME);
}

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple demo of a kernel thread.");

module_init(hello_init);
module_exit(hello_exit);

#
# make && rmmod kt1 ; insmod kt1.ko ; dmesg|tail -n15
```

```
make -C /lib/modules/2.6.24-16-generic/build M=/<...>/kthreads modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.24-16-generic'
  CC [M]  /<...>/kthreads/kt1.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /<...>/kthreads/kt1.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.24-16-generic'
ERROR: Module kt1 does not exist in /proc/modules
[ 5502.052660] hello_init:44 : Create a kernel thread...
[ 5502.052673] hello_init:52 : Module kt initialized, thread PID 22294 alive.
[ 5502.052699] mykt:19 : name: mykt PID: 22294 TGID: 22294
[ 5502.052701] mykt:21 : mm field NULL.
[ 5502.052703] mykt:27 : K Thread 22294 going to sleep now...
# ps ax|grep '\[mykt\]'
22294 ?        S      0:00 [mykt]
# ps -Al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
--snip--
1 S     0 22294     2  0  80   0 -     0 -      ?        00:00:00 mykt
--snip--
#

Who's the parent of the new-born kernel thread? It's PID 2:
 2 ?        S<     0:00 [kthreadd]


# cat /proc/22294/status
Name:   mykt
State:  S (sleeping)
Tgid:   22294
Pid:    22294
PPid:   2
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups: 0
Threads:        1
SigQ:   0/16374
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: fffffffffffffff9
SigIgn: 0000000000000000
SigCgt: 0000000000000006
CapInh: 0000000000000000
CapPrm: 00000000ffffffff
CapEff: 00000000fffffeff
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     0
# cat /proc/22294/maps
#                                       ← Look Ma, no VMAs!

Lets signal our kernel thread now...
```

```
# kill -9 22294
# ps ax|grep '\[mykt\]'
22294 ?        S      0:00 [mykt]  ←  kill -9 has no effect as SIGKILL
                                      is blocked.
#
# kill -2 22294                        ← send SIGINT
# ps ax|grep '\[mykt\]'
# dmesg |tail
[ 5502.052660] hello_init:44 : Create a kernel thread...
[ 5502.052673] hello_init:52 : Module kt initialized, thread PID 22294 alive.
[ 5502.052699] mykt:19 : name: mykt PID: 22294 TGID: 22294
[ 5502.052701] mykt:21 : mm field NULL.
[ 5502.052703] mykt:27 : K Thread 22294 going to sleep now...
[ 6122.253969] mykt:36 : K thread 22294 exiting now...
#
```

Caution: The kernel thread used in this example is created by a kernel module. There exists a dangerous race – if one unloads the kernel module (with rmmod) and then signals the thread – the kernel will crash (an Oops! will occour), as the code/data was already unloaded.

We can do this (and thus fix the bug), by calling kthread_stop() in the exit routine.

```
int kthread_stop(struct task_struct *k);
```

But here, we don't have a task_struct ptr, hence cannot invoke this routine. (Of course, we could manually search through the task list for the matching pid; but this is O(n) and quite ugly).

The solution:
Use the API *kthread_run()*. It returns pointer to task_struct .

Note:-

A wrapper API to create, internally daemonize (setup) and start execution of a kernel thread is *kthread_run().*

```
From include/linux/kthread.h
/* … */

/**
 * kthread_run - create and wake a thread.
 * @threadfn: the function to run until signal_pending(current).
 * @data: data ptr for @threadfn.
```

```
 * @namefmt: printf-style name for the thread.
 *
 * Description: Convenient wrapper for kthread_create() followed by
 * wake_up_process().  Returns the kthread or ERR_PTR(-ENOMEM).
 */
#define kthread_run(threadfn, data, namefmt, ...)             \
({                                          \
    struct task_struct *__k                         \
        = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
    if (!IS_ERR(__k))                           \
        wake_up_process(__k);                       \
    __k;                                    \
})
```

Usage examples:

- *The USB Mass Storage driver implementation uses this API to implement a control thread:*

drivers/usb/storage/usb.c :

```
/* … */

    /* Start up our control thread */
    th = kthread_run(usb_stor_control_thread, us, "usb-storage");
    if (IS_ERR(th)) {
/* … */
```

- The modern IRQ handling code in *kernel/irq/manage.c*:
  `request_irq() → request_threaded_irq → __setup_irq()`

```
…
static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
...
 /*
     * Create a handler thread when a thread function is supplied
     * and the interrupt does not nest into another interrupt
     * thread.
     */
    if (new->thread_fn && !nested) {
        struct task_struct *t;

        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                   new->name);
        if (IS_ERR(t)) {
            ret = PTR_ERR(t);
            goto out_mput;
          }
```

. . .

- Also, see the 'kt2' demo kernel module for usage of kthread_run(), etc (provided with participant courseware).

Work Queues

Work queues are a different form of deferring work from what we have looked at so far. Work queues defer work into a kernel thread - this bottom half always runs in process context. Thus, code deferred to a work queue has all the usual benefits of process context. Most importantly, work queues are schedulable and can therefore sleep.

Normally, it is easy to decide between using work queues and softirqs/tasklets. If the *deferred work needs to sleep, work queues are used*. If the deferred work need not sleep, softirqs or tasklets are used. Indeed, the usual alternative to work queues is kernel threads. Because the kernel developers frown upon creating a new kernel thread (and, in some locales, it is a punishable offense), work queues are strongly preferred. They are really easy to use, too.

If you need a schedulable entity to perform your bottom-half processing, you need work queues. They are the only bottom-half mechanisms that run in process context, and thus, the only ones that can sleep. This means they are useful for situations where you need to allocate a lot of memory, obtain a semaphore, or perform block I/O. If you do not need a kernel thread to handle your deferred work, consider a tasklet instead.

--snip--

Using Work Queues

Using work queues is easy. We cover the default events queue first, and then look at creating new worker threads.

Creating Work

The first step is actually creating some work to defer. To create the structure statically at run-time, use `DECLARE_WORK`:

Note 1-
Newer kernels *don't supply a third parameter* to INIT_WORK and DECLARE_WORK (the void *data).

```
DECLARE_WORK(name, function);
```

Example of using DECLARE_WORK in the kernel:

```
static DECLARE_WORK(cad_work, deferred_cad);

        if (C_A_D)   << the 3 finger salute: Ctrl_Alt_Del :) >>
                schedule_work(&cad_work);
```

So, how can we pass data to a workqueue??
See this post on stackoverflow.

Alternatively, you can create work at run-time via a pointer:

```
INIT_WORK(struct work_struct *work,
          void (*func)(struct work_struct *));
```

This dynamically initializes the work queue pointed to by work with handler function func.

<<
Older:

```
DECLARE_WORK(name, void (*func)(void *), void *data);
```

This statically creates a work_struct structure named name with handler function func and argument data.
>>

<<

Note 2 -

One way to see runtime usage of workqueues is to use the trace-cmd front-end to the ftrace facility:

```
# trace-cmd record -e workqueue
Hit Ctrl^C to stop recording

^C
```

```
...
# trace-cmd report
…
cpus=8
          <idle>-0     [005]  7995.171325: workqueue_queue_work: work
struct=0xffffffff81a64360 function=console_callback workqueue=0xffff8801376eb000
req_cpu=5 cpu=5
          <idle>-0     [005]  7995.171327: workqueue_activate_work: work struct
0xffffffff81a64360
          <idle>-0     [005]  7995.171336: workqueue_queue_work: work
struct=0xffff880133e5c110 function=flush_to_ldisc workqueue=0xffff8801376eb000
req_cpu=5 cpu=5
          <idle>-0     [005]  7995.171336: workqueue_activate_work: work struct
0xffff880133e5c110
     kworker/5:1-47    [005]  7995.171409: workqueue_execute_start: work struct
0xffffffff81a64360: function console_callback
     kworker/5:1-47    [005]  7995.171413: workqueue_execute_end: work struct
0xffffffff81a64360
     kworker/5:1-47    [005]  7995.171414: workqueue_execute_start: work struct
0xffff880133e5c110: function flush_to_ldisc
...
          <idle>-0     [004]  7995.175836: workqueue_queue_work: work
struct=0xffff8800bb311188 function=do_dbs_timer workqueue=0xffff880134e5da80
req_cpu=4 cpu=4
          <idle>-0     [000]  7995.175836: workqueue_queue_work: work
struct=0xffff8800bb211188 function=do_dbs_timer workqueue=0xffff880134e5da80
req_cpu=0 cpu=0
…
…
      <idle>-0     [001]  7998.694408: workqueue_activate_work: work struct
0xffff88009a8cc110
     kworker/1:0-8     [001]  7998.694420: workqueue_execute_start: work struct
0xffff8800bb251188: function do_dbs_timer
     kworker/1:0-8     [001]  7998.694424: workqueue_execute_end: work struct
0xffff8800bb251188
     kworker/1:0-8     [001]  7998.694425: workqueue_execute_start: work struct
0xffff88009a8cc110: function flush_to_ldisc
     kworker/1:0-8     [001]  7998.694471: workqueue_execute_end: work struct
0xffff88009a8cc110
#

>>
```

Your Work Queue Handler

The prototype for the work queue handler is

```
void work_handler(struct work_struct *)

[ OLDER : void work_handler(void *data) ]
```

A worker thread executes this function, and thus, the function runs in process context. By

default, interrupts are enabled and no locks are held. If needed, the function can sleep. Note that, despite running in process context, the work handlers cannot access user-space memory because there is no associated user-space memory map for kernel threads. The kernel can access user memory only when running on behalf of a user-space process, such as when executing a system call. Only then is user memory mapped in.

Locking between work queues or other parts of the kernel is handled just as with any other process context code. This makes writing work handlers much easier. The next two chapters cover locking.

Scheduling Work

Now that the work is created, we can schedule it. To queue a given work's handler function with the default events worker threads, simply call

```
schedule_work(&work);
```

The work is scheduled immediately and is run as soon as the events worker thread on the current processor wakes up.

include/linux/workqueue.h
```
/**
 * schedule_work - put work task in global workqueue
 * @work: job to be done
 *
 * Returns %false if @work was already on the kernel-global workqueue and
 * %true otherwise.
 *
 * This puts a job in the kernel-global workqueue if it was not already
 * queued and leaves it in the same position on the kernel-global
 * workqueue otherwise.
 */
static inline bool schedule_work(struct work_struct *work)
{
    return queue_work(system_wq, work);
}
```

Sometimes you do not want the work to execute immediately, but instead after some delay. In those cases, you can schedule work to execute at a given time in the future:

```
schedule_delayed_work(&work, delay);
```

In this case, the `work_struct` represented by `&work` will not execute for at least delay timer ticks into the future. Using ticks as a unit of time is covered in Chapter 10.

```
/**
 * schedule_delayed_work - put work task in global workqueue after delay
 * @dwork: job to be done
 * @delay: number of jiffies to wait or 0 for immediate execution
 *
 * After waiting for a given time this puts a job in the kernel-global
 * workqueue.
 */
static inline bool schedule_delayed_work(struct delayed_work *dwork,
                                         unsigned long delay)
{
    return queue_delayed_work(system_wq, dwork, delay);
}
```

<<

The default kernel thread(s) that consume your work queue are a set of n threads – one per processor core – called “events/n” where n is the CPU number.

```
# ps -Aef|grep 'events/[0-9]'
root         9     2  0 14:45 ?        00:00:00 [events/0]
root     10461     2  0 18:59 ?        00:00:00 [events/1]
#
```

Creating the global workqueues is done here:

kernel/workqueue.c

```
...
    system_wq = alloc_workqueue("events", 0, 0);
    system_highpri_wq = alloc_workqueue("events_highpri", WQ_HIGHPRI, 0);
    system_long_wq = alloc_workqueue("events_long", 0, 0);
    system_unbound_wq = alloc_workqueue("events_unbound", WQ_UNBOUND,
                        WQ_UNBOUND_MAX_ACTIVE);
    system_freezable_wq = alloc_workqueue("events_freezable",
                          WQ_FREEZABLE, 0);
    system_power_efficient_wq = alloc_workqueue("events_power_efficient",
                          WQ_POWER_EFFICIENT, 0);
    system_freezable_power_efficient_wq = alloc_workqueue("events_freezable_power_efficient",
                          WQ_FREEZABLE | WQ_POWER_EFFICIENT,
                          0);
    BUG_ON(!system_wq || !system_highpri_wq || !system_long_wq ||
           !system_unbound_wq || !system_freezable_wq ||
           !system_power_efficient_wq ||
           !system_freezable_power_efficient_wq);
...
```

>>

<<

See the Broadcom 'Blutonium' firmware driver as a simple example of using workqueues: *drivers/bluetooth/bcm203x.c*

>>

Flushing Work

Queued work is executed when the worker thread next wakes up. Sometimes, you need to ensure that a given batch of work has completed before continuing. This is especially important for modules, which almost certainly want to call this function before unloading. Other places in the kernel also might need to make certain no work is pending, to prevent race conditions.

For these needs, there is a function to flush a given work queue:

```
void flush_scheduled_work(void);
```

This function waits until all entries in the queue are executed before returning. While waiting for any pending work to execute, the function sleeps. Therefore, you can call it only from process context.

Note that this function does not cancel any delayed work. That is, any work that was scheduled via schedule_delayed_work(), and whose delay is not yet up, is not flushed via flush_scheduled_work(). To cancel delayed work, call

```
int cancel_delayed_work(struct work_struct *work);
```

This function cancels the pending work, if any, associated with the given work_struct.

Creating New Work Queues

If the default queue is insufficient for your needs, you can create a new work queue and corresponding worker threads. Because this creates one worker thread per processor, you should create unique work queues only if your code really needs the performance of a unique set of threads.You create a new work queue and the associated worker threads via a simple function:

```
struct workqueue_struct *create_workqueue(const char *name);
```

The parameter name is used to name the kernel threads. For example, the default events queue is created via

```
struct workqueue_struct *keventd_wq;
```

```
keventd_wq = create_workqueue("events");
```

```
<<
# ps -Aef|grep 'events/[0-9]'
root         9     2  0 14:45 ?        00:00:00 [events/0]
root     10461     2  0 18:59 ?        00:00:00 [events/1]
#
>>
```

This function creates all the worker threads (one for each processor in the system) and prepares them to handle work.

Creating work is handled in the same manner regardless of the queue type. After the work is created, the following functions are analogous to schedule_work() and schedule_delayed_work(), except that they work on the given work queue and not the default events queue.

```
int queue_work(struct workqueue_struct *wq,
                            struct work_struct *work)

int queue_delayed_work(struct workqueue_struct *wq,
                       struct work_struct *work,
                       unsigned long delay)
```

Finally, you can flush a wait queue via a call to the function `flush_workqueue(struct workqueue_struct *wq) .`

As previously discussed, this function works identically to flush_scheduled_work(), except that it waits for the given queue to empty before returning.

Code Walkthrough Session:

Using Work Queues – a simple demo:
Participants can find the source code of the "slpy_wq" driver on the accompanying CD.

```
$ sudo /bin/bash
Password:
#
# insmod slpy_wq.ko
# alias dm
alias dm='dmesg|tail -n35'
#
# dm
[ 1234.539440] sleepy: major # = 252
[ 1234.539447] slpy_init_module:154 : Loaded ok.
```

```
# ./rd_tst sleepy &          << Put some processes to sleep on the WQ >>
[1] 12245
# device opened: fd=3

# ./rd_tst sleepy &
device opened: fd=3
[2] 12250
# ./rd_tst sleepy &
[3] 12251
device opened: fd=3
# ps
  PID TTY          TIME CMD
 9532 pts/0    00:00:00 bash
12245 pts/0    00:00:00 rd_tst
12250 pts/0    00:00:00 rd_tst
12251 pts/0    00:00:00 rd_tst
12253 pts/0    00:00:00 ps
# dm
[ 1234.539440] sleepy: major # = 252
[ 1234.539447] slpy_init_module:154 : Loaded ok.
[ 1251.721504] sleepy_read:102 : process 12245 (rd_tst) going to sleep
[ 1253.162118] sleepy_read:102 : process 12250 (rd_tst) going to sleep
[ 1254.556975] sleepy_read:102 : process 12251 (rd_tst) going to sleep
#
                    << We can see that 3 processes are asleep on the WQ >>

# ./wr_tst sleepy       << Wake them up, and trigger the scheduling of the
                                Work Queue >>
sleepy:: wrote 1024 bytes
sleepy [12251]:: read 0 bytes: buf is:
[3]+  Done                    ./rd_tst sleepy
sleepy [12245]:: read 0 bytes: buf is:
sleepy [12250]:: read 0 bytes: buf is:
#
[1]-  Done                    ./rd_tst sleepy
[2]+  Done                    ./rd_tst sleepy
# dm
[ 1234.539440] sleepy: major # = 252
[ 1234.539447] slpy_init_module:154 : Loaded ok.
[ 1251.721504] sleepy_read:102 : process 12245 (rd_tst) going to sleep
[ 1253.162118] sleepy_read:102 : process 12250 (rd_tst) going to sleep
[ 1254.556975] sleepy_read:102 : process 12251 (rd_tst) going to sleep
[ 1314.602126] sleepy_write:118 : Scheduling deferred work in the work queue now...
[ 1314.602139] sleepy_write:122 : process 12381 (wr_tst) awakening the readers...
[ 1314.602283] wq_func:88 : We're here doing some deferred work! jiffies=873429
[ 1314.602288] wq_func:89 : Now setting up a timer...
[ 1314.602298] sleepy_read:110 : awoken 12251 (rd_tst), data_present=1
[ 1314.602257] sleepy_read:110 : awoken 12245 (rd_tst), data_present=1
[ 1314.602907] sleepy_read:110 : awoken 12250 (rd_tst), data_present=1
#

                                    << Let a few seconds elapse... >>
```

```
# dmesg
[ 1234.539440] sleepy: major # = 252
[ 1234.539447] slpy_init_module:154 : Loaded ok.
[ 1251.721504] sleepy_read:102 : process 12245 (rd_tst) going to sleep
[ 1253.162118] sleepy_read:102 : process 12250 (rd_tst) going to sleep
[ 1254.556975] sleepy_read:102 : process 12251 (rd_tst) going to sleep
[ 1314.602126] sleepy_write:118 : Scheduling deferred work in the work queue now...
[ 1314.602139] sleepy_write:122 : process 12381 (wr_tst) awakening the readers...
[ 1314.602283] wq_func:88 : We're here doing some deferred work! jiffies=873429
[ 1314.602288] wq_func:89 : Now setting up a timer...
[ 1314.602298] sleepy_read:110 : awoken 12251 (rd_tst), data_present=1
[ 1314.602257] sleepy_read:110 : awoken 12245 (rd_tst), data_present=1
[ 1314.602907] sleepy_read:110 : awoken 12250 (rd_tst), data_present=1
[ 1321.583255] whee:73 : Timed out. count=1
# dmesg
...
[ 1314.602288] wq_func:89 : Now setting up a timer...
[ 1314.602298] sleepy_read:110 : awoken 12251 (rd_tst), data_present=1
[ 1314.602257] sleepy_read:110 : awoken 12245 (rd_tst), data_present=1
[ 1314.602907] sleepy_read:110 : awoken 12250 (rd_tst), data_present=1
[ 1321.583255] whee:73 : Timed out. count=1
[ 1328.567620] whee:73 : Timed out. count=2
[ 1335.551987] whee:73 : Timed out. count=3
#
                    << We can see that the timer has expired several times >>
# rmmod slpy_wq
# dmesg
...
[ 1321.583255] whee:73 : Timed out. count=1
[ 1328.567620] whee:73 : Timed out. count=2
[ 1335.551987] whee:73 : Timed out. count=3
[ 1342.536366] whee:73 : Timed out. count=4
[ 1347.408688] slpy_cleanup_module:163 : Removed.
#
```

If we enable the stack dump in the timeout function `whee()` , you would see output similar to the one shown below:

```
--snip--
[ 1892.120622] wq_func:88 : We're here doing some deferred work! jiffies=1018132
[ 1892.120627] wq_func:89 : Now setting up a timer...
[ 1892.120637] sleepy_read:110 : awoken 13181 (rd_tst), data_present=1
[ 1899.102866] whee:73 : Timed out. count=1
[ 1899.102877] Pid: 4, comm: ksoftirqd/0 Not tainted 2.6.24-16-generic #1
[ 1899.102907]  [<f8c7d27d>] whee+0x2d/0x80 [slpy_wq]
[ 1899.102924]  [<c0135909>] run_timer_softirq+0x169/0x1e0
[ 1899.102937]  [<f8c7d250>] whee+0x0/0x80 [slpy_wq]
[ 1899.102954]  [<f8c7d250>] whee+0x0/0x80 [slpy_wq]
[ 1899.102970]  [<c0131a22>] __do_softirq+0x82/0x110
[ 1899.102988]  [<c0131b05>] do_softirq+0x55/0x60
[ 1899.103000]  [<c0131e89>] ksoftirqd+0x89/0x100
[ 1899.103013]  [<c0131e00>] ksoftirqd+0x0/0x100
[ 1899.103025]  [<c01408b2>] kthread+0x42/0x70
```

```
[ 1899.103033]  [<c0140870>] kthread+0x0/0x70
[ 1899.103045]  [<c0105677>] kernel_thread_helper+0x7/0x10
[ 1899.103063]  =====================
--snip--
```

Invoking userspace applications from within the kernel

Use the *call_usermode_helper()* API.

```
extern int
call_usermodehelper(char *path, char **argv, char **envp, int wait);
```

Eg.
kernel/reboot.c

```
...
char poweroff_cmd[POWEROFF_CMD_PATH_LEN] = "/sbin/poweroff";
static const char reboot_cmd[] = "/sbin/reboot";

static int run_cmd(const char *cmd)
{
    char **argv;
    static char *envp[] = {
        "HOME=/",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin",
        NULL
    };
    int ret;
    argv = argv_split(GFP_KERNEL, cmd, NULL);
    if (argv) {
        ret = call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
        argv_free(argv);
    } else {
        ret = -ENOMEM;
    }

    return ret;
}
…

<<
#define UMH_NO_WAIT 0   /* don't wait at all */
#define UMH_WAIT_EXEC   1   /* wait for the exec, but not the process */
#define UMH_WAIT_PROC   2   /* wait for the process to complete */
#define UMH_KILLABLE    4   /* wait for EXEC/PROC killable */
>>
```

Summary :: Deferred Functionality Mechanisms in the 2.6 Linux Kernel

--snip--
You have seen the differences between interrupt handlers and bottom halves, but there are a few similarities, too. Interrupt handlers and tasklets are both not reentrant. And neither of them can go to sleep. Also, interrupt handlers, tasklets, and softirqs cannot be preempted.

Work queues are a third way to defer work from interrupt handlers. They execute in process context and are allowed to sleep, so they can use drowsy functions such as mutexes. We discussed work queues in the preceding chapter when we looked at various kernel helper facilities. Table 4.1 compares softirqs, tasklets, and work queues.

Table 4.1. Comparing Softirqs, Tasklets, and Work Queues

	Softirqs	Tasklets	Work Queues
Execution context	Deferred work runs in interrupt context.	Deferred work runs in interrupt context.	Deferred work runs in process context.
Reentrancy	Can run simultaneously on different CPUs.	Cannot run simultaneously on different CPUs. Different CPUs can run different tasklets, however.	Can run simultaneously on different CPUs.
Sleep semantics	Cannot go to sleep.	Cannot go to sleep.	May go to sleep.
Preemption	Cannot be preempted/scheduled.	Cannot be preempted/scheduled.	May be preempted/scheduled.
Ease of use	Not easy to use.	Easy to use.	Easy to use.

	Softirqs	Tasklets	Work Queues
When to use	If deferred work will not go to sleep and if you have crucial scalability or speed requirements.	If deferred work will not go to sleep.	If deferred work may go to sleep.

There is an ongoing debate in LKML on the feasibility of getting rid of the tasklet interface. Tasklets enjoy more priority than process context code, so they present latency problems. Moreover, as you learned, they are constrained not to sleep and to execute on the same CPU. It's being suggested that all existing tasklets be converted to softirqs or work queues on a case-by-case basis.
