Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Porting drivers to the 2.5 kernel

Jonathan Corbet
LWN.net
corbet@lwn.net

Abstract

The 2.5 development series has brought with it the usual large set of changes to the internal driver API. The end result is a kernel that is far more pleasant to program for, and a more robust and reliable system. The cost of all these changes, of course, is that kernel code—including device drivers—must be updated to work under the new regime. This paper will give an overview of what has changed in the internal kernel API, why the changes were made, and what must be done to make drivers work again. Some familiarity with kernel programming is assumed.

1 Introduction

2.5 was a busy time for kernel developers. Much work was done to make kernel code more reliable and less susceptible to common problems. There has also been a large emphasis on improved performance on high-end systems. The end result is that almost no part of the kernel—and almost no internal API—was left untouched. The degree of change varies from relatively small (for network drivers, for example) to extreme (block drivers). An awareness of these changes is helpful for anybody who is interested in how kernel development is proceeding, and crucial for anybody who must make code work with the new kernel.

This paper will start with the basic changes which affect all drivers—module loading, memory allocation, etc. Later sections will get into the more advanced topics, including the block layer, and memory management. Space constraints make it impossible to get into much detail here. A series of documents can be found at the web site listed at the end of this paper; those documents explore the topics found below in much greater depth, and will be kept current as the kernel evolves.

2 Loadable modules

The module loader was completely replaced in 2.5; the new implementation works almost entirely within the kernel. Interestingly, moving the module loader into the kernel resulted in a net reduction in kernel code. This development has forced a few changes in how modules work, however.

2.1 Hello world

The obvious place to start is the classic “hello world” program, which, in this context, is implemented as a kernel module. The 2.4 version of this module looked like:

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk(KERN_INFO "Hello, world\n");
    return 0;
}

void cleanup_module(void)
```

```
{
    printk(KERN_INFO "Goodbye cruel world\n");
}
```

One would not expect that something this simple and useless would require much in the way of changes, but, in fact, this module will not quite work in a 2.5 kernel. So what do we have to do to fix it up?

The first change is relatively insignificant; the first line:

```
#define MODULE
```

is no longer necessary, since the kernel build system defines it for you.

The biggest problem with this module, however, is that you have to explicitly declare your initialization and cleanup functions with `module_init` and `module_exit`, which are found in `<linux/init.h>`. You really should have done that for 2.4 as well, but you could get away without it as long as you used the names `init_module` and `cleanup_module`. You can still get away with it, but the new module code broke this way of doing things once, and could do so again. It's time to bite the bullet and do things right.

With these changes, "hello world" now looks like:

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

static int hello_init(void) {
  printk(KERN_ALERT "Hello, world\n");
  return 0;
}

static void hello_exit(void) {
  printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

This module will now work—the "Hello, world" message shows up in the system log file. At least, once you have succeeded in building the module properly...

2.2 Module compilation

One result of the module changes (combined with significant changed in the kernel build mechanism) is that compiling loadable modules has gotten a bit more complicated. In the 2.4 days, a makefile for an external module could be put together in just about any old way; the job of creating a loadable module was handled in a single, simple compilation step. All you really needed was a handy set of kernel headers to compile against.

With the 2.5 kernel, you still need those headers. You also, however, need a configured kernel source tree and a set of makefile rules describing how modules are built. All this is required because the new module loader needs some additional symbols defined at compilation time; because all modules must now go through a linking step (even single-file modules); and because the new modversions implementation requires a separate processing step.

One could certainly, with some effort, write a new, standalone makefile which would handle the above issues. But that solution, along with being a pain, is also brittle; as soon as the module build process changes again, the makefile will break. Eventually that process will stabilize, but, for a while, further changes are almost guaranteed.

So, now that you are convinced that you want to use the kernel build system for external modules, how is that to be done? The first step is to learn how kernel makefiles work in general; `makefiles.txt` from a recent kernel's `Documentation/kbuild` directory is recommended reading. The makefile magic needed for a simple kernel module is minimal, however. In fact, for a single-file module, a single-line makefile will suffice:

```
obj-m := module.o
```

(where `module` is replaced with the actual name of the resulting module, of course). The kernel build system, on seeing that declaration, will compile `module.o` from `module.c`, link it with `vermagic.o` from the kernel tree, and leave the result in `module.ko`, which can then be loaded into the kernel.

A multi-file module is almost as easy:

```
obj-m := module.o
module-objs := file1.o file2.o
```

In this case, `file1.c` and `file2.c` will be compiled, then linked into `module.ko`.

Of course, all this assumes that you can get the kernel build system to read and deal with your makefile. The magic command to make that happen is something like the following:

```
make -C /usr/src/linux \
        SUBDIRS=\$PWD modules
```

Where `/usr/src/linux` is the path to the source directory for the target kernel. This command causes `make` to head over to the kernel source to find the top-level makefile; it then moves back to the original directory to build the module of interest.

Of course, typing that command could get tiresome after a while. A trick posted by Gerd Knorr can make things a little easier, though. By looking for a symbol defined by the kernel build process, a makefile can determine whether it has been read directly, or by way of the kernel build system. So the following will build a module against the source for the currently running kernel:

```
ifneq ($(KERNELRELEASE),)
obj-m := module.o
else
KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)
default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

Now a simple “make” will suffice. The makefile will be read twice; the first time it will simply invoke the kernel build system, while the actual work will get done in the second pass. A makefile written in this way is simple, and it should be robust with regard to kernel build changes.

2.3 Module parameters

The old `MODULE_PARM` macro, which used to specify parameters which can be passed to the module at load time, is no more. The new parameter declaration scheme add type safety and new functionality, but at the cost of breaking compatibility with older modules.

Modules with parameters should now include `<linux/moduleparam.h>` explicitly. Parameters are then declared with `module_param`:

```
module_param(name, type, perm);
```

Where `name` is the name of the parameter (and of the variable holding its value), `type` is its type, and `perm` is the permissions to be applied to that parameter’s sysfs entry. The `type` parameter can be one of `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` or `invbool`. That type will be verified during compilation, so it is no longer possible to create confusion by declaring module parameters with mismatched types. The plan is for module parameters to appear automatically in sysfs, but that feature had not been implemented as of 2.5.69; for now, the safest alternative is to set `perm` to zero, which means “no sysfs entry.”

If the name of the parameter as seen outside the module differs from the name of the variable

used to hold the parameter's value, a variant on `module_param` may be used:

```
module_param_named(name, value, type, perm);
```

Where `name` is the externally-visible name and `value` is the internal variable.

String parameters will normally be declared with the `charp` type; the associated variable is a `char` pointer which will be set to the parameter's value. If you need to have a string value copied directly into a `char` array, declare it as:

```
module_param_string(name, string, len, perm);
```

Usually, `len` is best specified as `sizeof(string)`.

2.4 The module use count

In 2.4 and prior kernels, modules maintained their "use count" with macros like `MOD_INC_USE_COUNT`. The use count, of course, is intended to prevent modules from being unloaded while they are being used. This method was always somewhat error prone, especially when the use count was manipulated inside the module itself. In the 2.5 kernel, reference counting is handled differently.

The only safe way to manipulate the count of references to a module is outside of the module's code. Otherwise, there will always be times when the kernel is executing within the module, but the reference count is zero. So this work has been moved outside of the modules, and life is generally easier for module authors.

Any code which wishes to call into a module (or use some other module resource) must first attempt to increment that module's reference count:

```
int try_module_get(&module);
```

It is also necessary to look at the return value; a zero return means that the try failed (perhaps the module is being unloaded), and the module should not be used.

A reference to a module can be released with `module_put()`.

Again, modules will not normally have to manage their own reference counts. The only exception may be if a module provides a reference to an internal data structure or function that is not accounted for otherwise. In that (rare) case, a module could conceivably call `try_module_get()` on itself.

2.5 Exporting symbols

In 2.5, module symbols are not exported by default. Chances are that change will cause few problems. When you get a chance, however, you can remove `EXPORT_NO_SYMBOLS` lines from your module source. Exporting no symbols is now the default, so `EXPORT_NO_SYMBOLS` is a no-op.

The 2.4 `inter_module_` functions have been deprecated as unsafe. The `symbol_get()` function exists for the cases when normal symbol linking does not work well enough. Its use requires setting up weak references at compile time, and is beyond the scope of this document.

2.6 Kernel version checking

2.4 and prior kernels would include, in each module, a string containing the version of the kernel that the module was compiled against. Normally, modules would not be loaded if the compile version failed to match the running kernel.

In 2.5, things still work mostly that way. The kernel version is loaded into a separate, "link-once" ELF section, however, rather than be-

ing a visible variable within the module itself. As a result, multi-file modules no longer need to define `__NO_VERSION__` before including `<linux/module.h>`.

The new "version magic" scheme also records other information, including the compiler version, SMP status, and preempt status; it is thus able to catch more incompatible situations than the old scheme did.

3 The device model

One of the more significant changes in the 2.5 development series is the creation of the integrated device model. The device model was originally intended to make power management tasks easier through the maintenance of a representation of the host system's hardware structure. A certain amount of mission creep has occurred, however, and the device model is now closely tied into a number of device management tasks—and other kernel functions as well.

The device model presents a bit of a steep learning curve when first encountered. The fact that the whole thing is still (as of 2.5.69) in a state of fairly serious flux doesn't help, especially considering that the documentation is, in many cases, a few revisions behind the actual code. But the underlying concepts are not *that* hard to understand, and driver programmers will benefit from a grasp of what's going on.

The fundamental task of the driver model is to maintain a set of internal data structures which reflect the architecture and state of the underlying system. Among other things, the driver model tracks:

- Which devices exist in the system, what power state they are in, what bus they are attached to, and which driver is responsible for them.
- The bus structure of the system; which buses are connected to which others (i.e., a USB controller can be plugged into a PCI bus), and which devices each bus can potentially support (along with associated drivers), and which devices actually exist.
- The device drivers known to the system, which devices they can support, and which bus type they know about.
- What *kinds* of devices ("classes") exist, and which real devices of each class are connected. The driver model can thus answer questions like "where is the mouse (or mice) on this system?" without the need to worry about how the mouse might be physically connected.
- And many other things.

Underneath it all, the driver model works by tracking system configuration changes (hardware and software) and maintaining a complex "web woven by a spider on drugs" data structure to represent it all.

Many driver programmers will be able to get away with ignoring the device model altogether; most of the gory details are handled at the bus level. There are times, however, when an understanding of what's going on can be useful. A full discussion of the device model would require a talk of its own (indeed, there are two on the OLS schedule); suffice to say, for now, that details can be found on the web site.

4 Support interfaces

Now that we know how to compile a module, it's time to look at the various other changes it

will need to be adapted for. We'll start with various low-level support interfaces used by many or most drivers (and other modules).

4.1 Memory allocation

The 2.5 development series has brought relatively few changes to the way device drivers will allocate and manage memory. In fact, most drivers should work with no changes in this regard. There are a few improvements that have been made, however, that are worth a mention. These include some changes to page allocation, and the new "mempool" interface.

4.1.1 Allocation flags

The old `<linux/malloc.h>` include file is gone; it is now necessary to include `<linux/slab.h>` instead.

The `GFP_BUFFER` allocation flag is gone (it was actually removed in 2.4.6). That will bother few people, since almost nobody used it. For reference, here is the full set of 2.5 allocation flags, from the most restrictive to the least:

`GFP_ATOMIC`: a high-priority allocation which will not sleep; this is the flag to use in interrupt handlers and other non-blocking situations.

`GFP_NOIO`: blocking is possible, but no I/O will be performed.

`GFP_NOFS`: no filesystem operations will be performed.

`GFP_KERNEL`: a regular, blocking allocation.

`GFP_USER`: a blocking allocation for user-space pages.

`GFP_HIGHUSER`: for allocating user-space pages where high memory may be used.

The `__GFP_DMA` and `__GFP_HIGHMEM` flags still exist and may be added to the above to direct an allocation to a particular memory zone. In addition, 2.5.69 added some new modifiers:

`__GFP_REPEAT`: This flag tells the page allocater to "try harder," repeating failed allocation attempts if need be. Allocations can still fail, but failure should be less likely.

`__GFP_NOFAIL`: Try even harder; allocations with this flag must not fail. Needless to say, such an allocation could take a long time to satisfy.

`__GFP_NORETRY`: Failed allocations should not be retried; instead, a failure status will be returned to the caller immediately.

The `__GFP_NOFAIL` flag is sure to be tempting to programmers who would rather not code failure paths, but that temptation should be resisted most of the time. Only allocations which truly cannot be allowed to fail should use this flag.

4.1.2 Page-level allocation

For page-level allocations, the `alloc_pages()` and `get_free_page()` functions (and variants) exist as always. They are now defined in `<linux/gfp.h>`. There are a few new ones as well. On NUMA systems, the allocator will do its best to allocate pages on the same node as the caller. To explicitly allocate pages on a different NUMA node, use:

```
struct page *
alloc_pages_node(int node_id,
                 unsigned int gfp_mask,
                 unsigned int order);
```

4.1.3 vmalloc_to_page()

Occasionally, it is necessary to find a `struct page` pointer for a page obtained from `vmalloc()`; usually this need arises in the implementation of `nopage()` methods. In the past, a driver had to walk through the page tables to find this pointer. As of 2.5.5 (and 2.4.19), however, all that is needed is a call to:

```
struct page *vmalloc_to_page(void *address);
```

This call is not a variant of `vmalloc()`—it allocates no memory. It simply returns a pointer to the `struct page` associated with an address obtained from `vmalloc()`.

4.1.4 Memory pools

Memory pools were one of the very first changes in the 2.5 series—they were added to 2.5.1 to support the new block I/O layer. The purpose of mempools is to help out in situations where a memory allocation must succeed, but sleeping is not an option. To that end, mempools pre-allocate a pool of memory and reserve it until it is needed. Mempools make life easier in some situations, but they should be used with restraint; each mempool takes a chunk of kernel memory out of circulation and raises the minimum amount of memory the kernel needs to run effectively.

A full discussion of mempools doesn't fit into this document; see the web site for details on their use.

4.2 Per-CPU variables

The 2.5 kernel makes extensive use of per-CPU data—arrays containing one object for each processor on the system. Per-CPU variables are not suitable for every task, but, in situations where they can be used, they do offer a couple of advantages:

- Per-CPU variables have fewer locking requirements since they are (normally) only accessed by a single processor.
- Restricting each processor to its own area eliminates cache line bouncing and improves performance.

Examples of per-CPU data in the 2.5 kernel include lists of buffer heads, lists of hot and cold pages, various kernel and networking statistics (which are occasionally summed together into the full system values), timer queues, and so on. There are currently no drivers using per-CPU values, but some applications (i.e., networking statistics for high-bandwidth adapters) might benefit from their use. See the web site listed at the end of this paper for a full description of how to use per-CPU data.

4.3 Timekeeping

One might be tempted to think that the basic task of keeping track of the time would not change that much from one kernel to the next. And, in fact, most kernel code which worries about times (and time intervals) will likely work unchanged in the 2.5 series. Code which gets into the details of how the kernel manages time may well need to adapt to some changes, however.

4.3.1 Internal clock frequency

One change which *shouldn't* be problematic for most code is the change in the internal clock rate on the x86 architecture. In previous kernels, `HZ` was 100; in 2.5 it has been bumped up to 1000. If your code makes any assumptions about what `HZ` really was (or, by extension, what `jiffies` really signified), you may have to make some changes now.

4.3.2 Kernel time variables

With a 1KHz clock, a 32-bit `jiffies` will overflow in just under 50 days, leading to occasional problems. So the 2.5 kernel has a new counter called `jiffies_64`. With 64 bits to work with, `jiffies_64` will not wrap around in a time frame that need concern most of us—at least until some future kernel starts using a petahertz internal clock.

For what it's worth, on most architectures, the classic, 32-bit `jiffies` variable is now just the least significant half of `jiffies_64`. Note that, on 32-bit systems, a 64-bit `jiffies` value raises concurrency issues. It is deliberately not declared as a `volatile` value (for performance reasons), so the possibility exists that code like:

```
u64 my_time = jiffies_64;
```

could get an inconsistent version of the variable, where the top and bottom halves do not match. To avoid this possibility, code accessing `jiffies_64` should use `xtime_lock`, which is the new seqlock type as of 2.5.60. In most cases, though, it will be easier to just use the convenience function provided by the kernel:

```
#include <linux/jiffies.h>
u64 my_time = get_jiffies_64();
```

Users of the internal `xtime` variable will notice a couple of similar changes. One is that `xtime`, too, is now protected by `xtime_lock` (as it is in 2.4 as of 2.4.10), so any code which plays around with disabling interrupts or such before accessing `xtime` will need to change. The best solution is probably to use:

```
struct timespec current_kernel_time(void);
```

which takes care of locking for you. `xtime` also now is a `struct timespec` rather than `struct timeval`; the difference being that the sub-second part is called `tv_nsec`, and is in nanoseconds.

4.3.3 Timers

The kernel timer interface is essentially unchanged since 2.4, with one exception. The new function:

```
void add_timer_on(struct timer_list *timer,
                  int cpu);
```

will cause the timer function to run on the given CPU with the expiration time hits.

4.3.4 Delays

The 2.5 kernel includes a new macro `ndelay()`, which delays for a given number of nanoseconds. It can be useful for interactions with hardware which insists on very short delays between operations. On most architectures, however, `ndelay(n)` is equal to `udelay(1)` for waits of less than one microsecond.

4.4 Delayed tasks and workqueues

The longstanding task queue interface was removed in 2.5.41; in its place is a new "workqueue" mechanism. Workqueues are

very similar to task queues, but there are some important differences. Among other things, each workqueue has one or more dedicated worker threads (one per CPU) associated with it. So all tasks running out of workqueues have a process context, and can thus sleep. Note that access to user space is not possible from code running out of a workqueue; there simply is no user space to access. Drivers can create their own work queues—with their own worker threads—but there is a default queue (for each processor) provided by the kernel that will work in most situations.

See the web site for a detailed discussion of workqueues.

4.5 DMA support

The direct memory access (DMA) support layer has been extensively changed in 2.5, but, in many cases, device drivers should work unaltered. For developers working on new drivers, or for those wanting to keep their code current with the latest API, there are a fair number of changes to be aware of.

The most evident change is the creation of the new generic DMA layer. A new set of generic DMA functions has been added which is intended to provide a DMA support API that is not specific to any particular bus. The new functions look much like the older PCI-based ones; changing from one API to the other is a fairly automatic job. The full set of equivalences between old and new DMA functions may be found on the web site.

There has been one significant change in the creation of scatter/gather streaming DMA mappings. The 2.4 version of `struct scatterlist` used a `char *` pointer (called `address`) for the buffer to be mapped, with a `struct page` pointer that would be used only for high memory addresses. In 2.5, the `address` pointer is gone, and all scatterlists must be built using `struct page` pointers.

Other developments of interest here include support for DAC (64-bit) PCI DMA, an interface for explicitly non-coherent mappings, and PCI pools.

5 Kernel preemption

One significant change introduced in 2.5 is the preemptible kernel. Previously, a thread running in kernel space would run until it returned to user mode or voluntarily entered the scheduler. In 2.5, if preemption is configured in, kernel code can be interrupted at (almost) any time. As a result, the number of challenges relating to concurrency in the kernel goes up. But this is actually not that big a deal for code which was written to handle SMP properly—most of the time. If you have not yet gotten around to implementing proper locking for your 2.4 driver, kernel preemption should give you yet another reason to get that job done.

The preemptible kernel means that your driver code can be preempted whenever the scheduler decides there is something more important to do. “Something more important” could include re-entering your driver code in a different thread. There is one big, important exception, however: preemption will not happen if the currently-running code is holding a spinlock. Thus, the precautions which are taken to ensure mutual exclusion in the SMP environment also work with preemption. So most (properly written) code should work correctly under preemption with no changes.

That said, code which makes use of per-CPU variables should take extra care. A per-CPU variable may be safe from access by other processors, but preemption could create races on the same processor. Code using per-CPU vari-

ables should, if it is not already holding a spinlock, disable preemption if the possibility of concurrent access exists. Usually, macros like `get_cpu_var()` should be used for this purpose.

Should it be necessary to control preemption directly (something that should happen rarely), some macros in `<linux/preempt.h>` will be helpful. A call to `preempt_disable()` will keep preemption from happening, while `preempt_enable()` will make it possible again. If you want to re-enable preemption, but don't want to get preempted immediately (perhaps because you are about to finish up and reschedule anyway), `preempt_enable_no_resched()` is what you need.

One interesting side-effect of the preemption work is that it is now much easier to tell if a particular bit of kernel code is running within some sort of critical section. A single variable in the task structure now tracks the preemption, interrupt, and softirq states. A new macro, `in_atomic()`, tests all of these states and returns a nonzero value if the kernel is running code that should complete without interruption.

6 Sleeping and waiting

Contrary to expectations, the classic functions `sleep_on()` and `interruptible_sleep_on()` were not removed in the 2.5 series. It seems that they are still needed in a few places where (1) taking them out is quite a bit of work, and (2) they are actually used in a way that is safe. Most authors of kernel code should, however, pretend that those functions no longer exist. There are very few situations in which they can be used safely, and better alternatives exist.

6.1 Safe sleeping

Most of those alternatives have been around since 2.3 or earlier. In many situations, one can use the `wait_event()` macros:

```
DECLARE_WAIT_QUEUE_HEAD(queue);
wait_event(queue, condition);
int wait_event_interruptible (queue, condition);
```

These macros work as they did in 2.4: `condition` is a boolean condition which will be tested within the macro; the wait will end when the condition evaluates true. It is worth noting that these macros have moved from `<linux/sched.h>` to `<linux/wait.h>`, which seems a more sensible place for them. There is also a new one:

```
int wait_event_interruptible_timeout(
        queue, condition, timeout);
```

which will terminate the wait if the timeout expires.

In many situations, `wait_event()` does not provide enough flexibility—often because tricky locking is involved. The longstanding "manual sleep" method can be used in these cases. In 2.5, however, a set of helper functions has been added which makes this task easier. The modern equivalent of a manual sleep looks like:

```
DECLARE_WAIT_QUEUE_HEAD(queue);
DEFINE_WAIT(wait);

while (! condition) {
    prepare_to_wait(&queue, &wait,
                    TASK_INTERRUPTIBLE);
        if (! condition)
            schedule();
    finish_wait(&queue, &wait)
}
```

Use `prepare_to_wait_exclusive()` instead when an exclusive wait is needed. Note that the new macro `DEFINE_WAIT()` is used here, rather than `DECLARE_WAITQUEUE()`. The former should be used when the wait queue entry is to be used with `prepare_to_wait()`, and should

probably *not* be used in other situations unless you understand what it is doing (which we'll get into next).

6.2 Wait queue changes

In addition to being more concise and less error prone, using `prepare_to_wait()` can yield higher performance in situations where wakeups happen frequently. This improvement is obtained by causing the process to be removed from the wait queue immediately upon wakeup; that removal keeps the process from seeing multiple wakeups if it doesn't otherwise get around to removing itself for a bit.

The automatic wait queue removal is implemented via a change in the wait queue mechanism. Each wait queue entry now includes its own "wake function," whose job it is to handle wakeups. The default wake function (which has the surprising name `default_wake_function()`), behaves in the customary way: it sets the waiting task into the `TASK_RUNNING` state and handles scheduling issues. The `DEFINE_WAIT()` macro creates a wait queue entry with a different wake function, `autoremove_wake_function()`, which automatically takes the newly-awakened task out of the queue.

And that, of course, is how `DEFINE_WAIT()` differs from `DECLARE_WAITQUEUE()`—they set different wake functions. How the semantics of the two differ is not immediately evident from their names, but that's how it goes. (The new runtime initialization function `init_wait()` differs from the older `init_waitqueue_entry()` in exactly the same way).

If need be, you can define your own wake function—though the need for that should be quite rare (the only user, currently, is the support code for the `epoll()` system calls). See the web site for details on how this is done.

One other change that most programmers won't notice: a bunch of wait queue cruft from 2.4 (two different kinds of wait queue lock, wait queue debugging) has been removed from 2.5.

6.3 Completions

Completions are a simple synchronization mechanism that is preferable to sleeping and waking up in some situations. If you have a task that must simply sleep until some process has run its course, completions can do it easily and without race conditions. They are not strictly a 2.5 feature, having been added in 2.4.7, but they merit a quick summary here.

A completion is, essentially, a one-shot flag that says "things may proceed." Working with completions requires including `<linux/completion.h>` and creating a variable of type `struct completion`. This structure may be declared and initialized statically with:

```
DECLARE_COMPLETION(my_comp);
```

A dynamic initialization would look like:

```
struct completion my_comp;
init_completion(&my_comp);
```

When your driver begins some process whose completion must be waited for, it's simply a matter of passing your completion event to `wait_for_completion()`:

```
void
wait_for_completion(struct completion *comp);
```

When some other part of your code has decided that the completion has happened, it can wake up anybody who is waiting with one of:

```
void complete(struct completion *comp);
void complete_all(struct completion *comp);
```

The first form will wake up exactly one waiting process, while the second will wake up all processes waiting for that event. Note that completions are implemented in such a way that they will work properly even if `complete()` is called before `wait_for_completion()`.

If you do not use `complete_all()`, you should be able to use a completion structure multiple times without problem. It does not hurt, however, to reinitialize the structure before each use—so long as you do it before initiating the process that will call `complete()`! The macro `INIT_COMPLETION()` can be used to quickly reinitialize a completion structure that has been fully initialized at least once.

7 Interrupt handling

The kernel's handling of device interrupts has been massively reworked in the 2.5 series. Fortunately, very few of those changes are visible to the rest of the kernel; most well-written code should "just work" under 2.5. There are, however, two important exceptions: the return type of interrupt handlers has changed, and drivers which depend on being able to globally disable interrupts will require some changes for 2.5.

7.1 Interrupt handler return values

Prior to 2.5.69, interrupt handlers returned `void`. There is, however, one useful thing that interrupt handlers can tell the kernel: whether the interrupt was something they could handle or not. If a device starts generating spurious interrupts, the kernel would like to respond by blocking interrupts from that device. If no interrupt handler for a given IRQ has been registered, the kernel knows that any interrupt on that number is spurious. When interrupt handlers exist, however, they must tell the kernel about spurious interrupts.

So, interrupt handlers now return an `irqreturn_t` value; `void` handlers will no longer compile. If your interrupt handler recognizes and handles a given interrupt, it should return `IRQ_HANDLED`. If it knows that the interrupt was not on a device it manages, it can return `IRQ_NONE` instead. The macro `IRQ_RETVAL(handled)` can also be used; `handled` should be nonzero if the handler could deal with the interrupt. The "safe" value to return, if, for some reason you are not sure, is `IRQ_HANDLED`.

7.2 Disabling interrupts

In the 2.5 kernel, it is no longer possible to globally disable interrupts. In particular, the `cli()`, `sti()`, `save_flags()`, and `restore_flags()` functions are no longer available. Disabling interrupts across all processors in the system is simply no longer done. This behavior has been strongly discouraged for some time, so most code *should* have been converted by now.

The proper way to do this fixing, of course, is to figure out exactly which resources were being protected by disabling interrupts. Those resources can then be explicitly protected with spinlocks instead. The change is usually fairly straightforward, but it does require an understanding of what is really going on.

It is still possible to disable all interrupts locally with `local_save_flags()` or `local_irq_disable()`. A single interrupt can be disabled globally with `disable_irq()`. Some of the spinlock operations also disable interrupts on the local processor, of course.

8 Asynchronous I/O

One of the key "enterprise" features added to the 2.5 kernel is asynchronous I/O (AIO). The AIO facility allows user processes to initiate multiple I/O operations without waiting for any of them to complete; the status of the operations can then be retrieved at some later time. Block and network drivers are already fully asynchronous, and thus there is nothing special that needs to be done to them to support the new asynchronous operations. Character drivers, however, have a synchronous API, and will not support AIO without some additional work. For most char drivers, there is little benefit to be gained from AIO support. In a few rare cases, however, it may be beneficial to make AIO available to your users. The web site has details on how to do that.

9 Block drivers

The first big, disruptive changes to the 2.5 kernel came from the reworking of the block I/O layer. As 2.5 heads towards its final phases, the block layer is still seeing a lot of attention. As one might guess, the result of all this work is a great many changes as seen by driver authors—or anybody else who works with block I/O. The transition may be painful for some, but it's worth it: the new block layer is easier to work with and offers much better performance than its predecessor.

Space constraints make it impossible to cover all of the block layer changes here; they could easily justify an entire paper to themselves. These changes *are* covered in detail on the web site listed at the end of the paper. Here, however, we'll have to content ourselves with an overview.

So, what has changed with the block layer?

- A great deal of old cruft is gone. For example, it is no longer necessary to work with a whole set of global arrays within block drivers. These arrays (`blk_size`, `blksize_size`, `hardsect_size`, `read_ahead`, etc.) have simply vanished.
- As part of the cruft removal, most of the `<linux/blk.h>` macros (`DEVICE_NAME`, `DEVICE_NR`, `CURRENT`, `INIT_REQUEST`, etc.) have been removed, and the file itself should go away soon. It is still possible to implement a simple request loop for straightforward devices where performance is not a big issue, but the mechanisms have changed.
- The `io_request_lock` is gone; locking is now done on a per-queue basis.
- Request queues have, in general, gotten more sophisticated. There is simple support for tagged command queueing, along with features like request barriers and queue-time device command generation.
- Buffer heads are no longer used in the block layer; they have been replaced with the new "`bio`" structure. The new representation of block I/O operations is designed for flexibility and performance; it encourages keeping large operations intact. Simple drivers can pretend that the `bio` structure does not exist, but most performance-oriented drivers—i.e., those that want to implement clustering and DMA—will need to be changed to work with `bio`s.

  One of the most significant features of the `bio` structure is that it represents I/O buffers directly with `page` structures and offsets, not in terms of kernel virtual addresses. By default, I/O buffers can be located in high memory, on the assumption that computers equipped with that much

memory will also have reasonably modern I/O controllers. Support operations have been provided for tasks like `bio` splitting and the creation of DMA scatter/gather maps.

- Sector numbers can now be 64 bits wide, making it possible to support very large block devices.

- The rudimentary `gendisk` ("generic disk") structure from 2.4 has been greatly improved in 2.5; generic disks are now used extensively throughout the block layer. The most significant change for block driver authors may be the fact that partition handling has been moved up into the block layer, and drivers no longer need know anything about partitions. That is, of course, the way things should always have been.

The end result is a fair amount of short-term pain for maintainers of block drivers. It does not take long, however, to realize that the new interface is easier to program for and far more robust.

10 Network drivers

Here's the good news for people maintaining network drivers: once you have deal with the basic changes that affect all drivers and loadable modules, your driver will likely work as it is. The API that was presented to drivers in 2.4 is essentially unchanged. There has been no gratuitous network driver breakage in 2.5.

The bad news is: a bunch of useful new stuff has been added in 2.5. If you want your driver to use the features of 2.5 to get the best performance out of your hardware, you will have to spend some time dealing with changes.

10.1 NAPI

The most significant change, perhaps, is the addition of NAPI ("New API"), which is designed to improve the performance of high-speed networking. NAPI works through **interrupt mitigation** (reducing the thousands of interrupts per second that accompany high network traffic) and **packet throttling** (keeping packets out of the kernel if they will be dropped anyway). NAPI was also backported to the 2.4.20 kernel.

Converting drivers to NAPI is essentially a two-step process:

- Your driver should no longer process incoming packets in its interrupt handler. Instead, it should disable further "packet available" interrupts and tell the networking system to begin polling the interface.

- A new `poll()` method must be created which processes all available incoming packets (up to a kernel-specified limit). A new function, `netif_receive_skb()` has been set up to accept packets from `poll()` methods.

The web site has the inevitable details.

10.2 Receiving packets in non-interrupt mode

Network drivers tend to send packets into the kernel while running in interrupt mode. There are occasions where, instead, packets will be received by a driver running in process context. There is no problem with this mode of operation, but it is possible that the networking software interrupt which performs packet processing may be delayed, reducing performance. To avoid this problems, drivers handing packets to the kernel outside of interrupt context should use:

```
int netif_rx_ni(struct sk_buff *skb);
```

instead of `netif_rx()`.

10.3 Other 2.5 features

A number of other networking features were added in 2.5. Here is a quick summary of developments that driver developers may want to be aware of.

Ethtool support. Ethtool is a utility which can perform detailed configuration of network interfaces; it can be found on the gkernel SourceForge page. This tool can be used to query network information, tweak detailed operating parameters, control message logging, and more. Supporting ethtool requires implementing the `SIOCETHTOOL ioctl()` command, along with (parts of, at least) the lengthy set of ethtool commands. See `<linux/ethtool.h>` for a list of things that can be done. Implementing the message logging control features requires checking the logging settings before each `printk()` call; there is a set of convenience macros in `<linux/netdevice.h>` which make that checking a little easier.

VLAN support. The 2.5 kernel has support for 802.1q VLAN interfaces; this support has also been working its way into 2.4, with the core being merged in 2.4.14.

TCP segmentation offloading. The TSO feature can improve performance by offloading some TCP segmentation work to the adaptor and cutting back slightly on bus bandwidth. TSO is an advanced feature that can be tricky to implement with good performance; see the `tg3` or `e1000` drivers for examples of how it's done.

11 User space access

The `kiobuf` abstraction was introduced in 2.3 as a low-level way of representing I/O buffers. Its primary use, perhaps, was to represent zero-copy I/O operations going directly to or from user space. A number of problems were found with the `kiobuf` interface, however; among other things, it forced large I/O operations to be broken down into small chunks, and it was seen as a heavyweight data structure. So, in 2.5.43, kiobufs were removed from the kernel. Direct access to user space remains possible, however, as we'll see.

The modern equivalent of `map_user_kiobuf()` is a function called `get_user_pages()`:

```
int get_user_pages(
       struct task_struct *task,
       struct mm_struct *mm,
       unsigned long start,
       int len,
       int write,
       int force,
       struct page **pages,
       struct vm_area_struct **vmas);
```

`task` is the process performing the mapping; the primary purpose of this argument is to say who gets charged for page faults incurred while mapping the pages. This parameter is almost always passed as "`current`". The memory management structure for the user's address space is passed in the `mm` parameter; it is usually `current->mm`. Note that `get_user_pages()` expects that the caller will have a read lock on `mm->mmap_sem`. The `start` and `len` parameters describe the user-buffer to be mapped; `len` is in pages. If the memory will be written to, `write` should be non-zero. The `force` flag forces read or write access, even if the current page protection would otherwise not allow that access. The `pages` array (which should be big enough to hold `len` entries) will be filled with pointers to the `page` structures for the user pages. If `vmas` is non-`NULL`, it will be filled with a pointer to the `vm_area_struct` structure containing each page.

The return value is the number of pages actually mapped, or a negative error code if something goes wrong. Assuming things worked, the user pages will be present (and locked) in memory, and can be accessed by way of the `struct page` pointers. Be aware, of course, that some or all of the pages could be in high memory.

There is no equivalent `put_user_pages()` function, so callers of `get_user_pages()` must perform the cleanup themselves. There are two things that need to be done: marking of modified pages, and releasing them from the page cache. If your device modified the user pages, the virtual memory subsystem may not know about it, and may fail to write the pages to permanent storage (or swap). That, of course, could lead to data corruption and grumpy users. The way to avoid this problem is to call:

```
int set_page_dirty_lock(struct page *page);
```

for each page in the mapping.

Finally, every mapped page must be released from the page cache, or it will stay there forever; simply pass each page structure to:

```
void put_page(struct page *page);
```

After you have released the page, of course, you should not access it again.

For a good example of how to use `get_user_pages()` in a char driver, see the definition of `sgl_map_user_pages()` in `drivers/scsi/st.c`.

12 Conclusion

This paper is drawn from the LWN.net "Porting Drivers to 2.5" series, which can be found at:

`http://lwn.net/Articles/driver-porting/`

Those articles contain much more detail than was possible to squeeze into this paper. They are also being maintained as kernel development continues; there are, beyond a doubt, things that have changed since this paper was written. If nothing else, the kernel developers will eventually figure out how they want to support larger device numbers. The driver-porting web site will have the latest information on kernel API changes.

13 Acknowledgments

Previous versions of this material have been improved by comments from Jens Axboe, Jamal Hadi Salim, Greg Kroah-Hartman, Andrew Morton, Rusty Russell, and others I have certainly forgotten. The creation of the "driver porting" series of articles was funded by LWN.net subscribers.