Smallsample

May 2011
Minimal Kernel Modules for Linux-2.6

Alessandro Rubini (`rubini@gnudd.com`)

Introduction

This packages is my collection of minimal examples for kernel programming, plus some user-space utilities. I am testing it on all official kernel releases starting from 2.6.20, up to the version number of the package you downloaded (2.6.37 as I write this).

The code is meant to run without special hardware, and I tested it both on real hardware and *qemu*. No platform dependencies are there (endianness or such stuff) and the code has been tested on a few platforms.

1 Git Branches

Since version 2.0 (now replaced by the one you are reading) this package uses git and the whole git information is part of the distribution (subdirectory `.git/`). The master branch, already checked-out in the distributed tarball is for the latest supported version, but there are branches for older versions where code differs from the latest supported.

To understand and use information in this chapter you are expected to have git installed (package *git-core* in some distributions, command `/usr/bin/git`), otherwise skip to the next section.

To get the list of branches in this package, use:

```
git branch
```

You should see one branch for each supported version, plus one branch for each published releae of this package. To see the difference from one branch to the other you can use the various *git diff* or *git log* options. For example:

```
git diff master for-2.6.23
git log master..for-2.6.23
```

In general, branches for older versions have additional commits than branches for later versions, as I apply backwards fixes starting from the master branch. Therefore, the branch called (for example) `release-smallsample-2.6.36` includes all commits for that release, including all the backward patches. In this case it's two backward commits:

```
903a0eb... 2.6.22 (backwards): smallwork: use older cancel
f450d02... 2.6.26 (backwards): smallchar-udev: device_create fix
af2c2f0... doc: documented the 2 timing modules
7645ef4... test: added smallwork
[...]
```

Please note that while the master branch will always move forward, I'll rebase the other branches as code changes, in order to have new features or fixes available in all applicable versions – that's why I also have a branch for the release, it's for any user who made own patches on an older version in a previous release of this package, so all previous commits are still there in later releases of the package.

I've made my best to have all supported versions (2.6.20 onwards) compile without errors or warnings, and I have tested all modules on all versions. Anyways, I may have overlooked some detail: any feedback is welcome.

Please note that some errors or warnings may depend on the configuration you are running. For example, if your kernel source doesn't support the frame buffer you will have some missing symbols while compiling *smallfb*. This is not a problem as long as you don't load the affected modules.

2 Compiling

To compile the modules, you need to point the `LINUX` environment variable to the top-level directory of the kernel tree you are going to load the modules in. This is the usual requirement for compiling modules. Note that your distribution may have a special package to help you build modules for the kernel you are currently running, but I urge to recompile your kernel.

A simple *make defconfig* will most likely work for the PC (either running natively or hosted on *qemu*) and *make versatile_defconfig* will work for ARM-Linux running under *qemu*. After compiling your kernel, you can compile the modules by issuing

```
        make LINUX=/path/to/your-kernel-2.6.37
```

As an alternative, you can *export LINUX=/path/to/your-kernel-2.6.37* and run *make* without special parameters. For cross-compiling, the usual rules apply: set `CROSS_COMPILE` and `ARCH` in your environment.

All modules but the two *hello* ones are called `small`*`something`*. The reasons is to tell them from real modules in your system. You can run something like “`grep small /proc/modules`” to find if you forgot some of these things in the system.

3 The Makefile

The *Makefile* in this package is the standard one for out-of-tree kernel modules. It uses the *obj-m* variable to name which objects to build and builds them from the top-level kernel directory.

The only difference from standard makefiles is that it enters the *userspace* subdirectory for both the *all* and *clean* targets.

4 User-space

The package includes a few user-space tools, in the *userspace* subdirectory. They are small tools that may be useful in testing the modules.

The current list is made of the following programs:

mapper
wmapper

The former program uses *mmap* to access a file and prints its content to *stdout*. The latter tool writes to a file after reading from *stdin*. Both receive as arguments the file name, the offset and the size of the data transfer.

selread

The program reads a device (filename passed on the command line) based on `select`. The *read* system call is only called after *select* reports that the file descriptor is readable.

5 Testing

This package includes some minimal testing facilities, in the `test/` subdirectory. Note however that there is no documentation outside of the scripts themselves, as it is mainly stuff I wrote for my own use.

6 Hello Modules

The most simple module is the classic *hello* module.

6.1 hello.ko

When loaded, `hello.ko` prints the usual message. When unloaded it prints another message. The module does nothing else, but it shows the basic features of a module:

- Inclusion of `<linux/>` headers.
- Use of `__init` and `__exit` for functions.
- Use of `static` definition for all file-local functions.
- Use of *module_init* and *module_exit*.
- Use of `MODULE_LICENSE`.

Note that you may write an even simpler module: one that returns an error from the init function: this technique is useful to run some code in kernel context without the need to unload the module when you need to repeat the action. This technique is especially useful when module parameter are used; for example, I use one such module to access *msr* registers on the PC, or monitor a GPIO pin in a busy-loop for one second.

6.2 helloparm.ko

The modules is like *hello* above, but introduces use of module parameters. It takes two parameters: an integer number and a string. The number is used to repeat the hello message several times, and the string is used as goodbye exit. Example use:

```
insmod helloparm.ko repeat=5 goodbye="I am done"
```

Note that the type of the parameters are `int` and `charp`. It may looks strange that a C language keyword and another generic word can be used in the same context; the word in the *module_parm* macro is actually used by the preprocessor to build a longer word as name for a structure. Thus, neither `int` nor `charp` appear as words in the compiler input.

7 Char Drivers

Most simple device drivers are char devices. While in many situations a char driver is not the best or cleanest solution, they are pretty easy to implement. The package has three modules, with a similar user interface and slightly different features.

7.1 smallchar.ko

The modules registers a major number using the old kernel function. It uses major 126 because it is “reserved for local or experimental use” (see `Documentation/devices.txt`). This means that no real device uses such major number. The device owns all minor numbers within the major number, and can thus use them at will.

This driver doesn’t use the minor number, and just offers a writable buffer of 64 bytes. The buffer is filled by *write* and can be read back through *read*. It is never shortened, so if you write less data than you originally wrote, the trailing part of the previous content appears back.

The driver introduces the following concepts:

- *register_chrdev* (now deprecated, see the next section).
- The *file_operation* structure.
- The `THIS_MODULE` macro, used to keep reference counts.

- The *write* and *read* method.
- The `__user` marker.
- `copy_to_user` and `copy_from_user`.
- Using the file offset and updating it.
- Returning errors to user space.

To use the driver, you must manually create the device special file, which customarily lives in `/dev`:

```
mknod /dev/sc c 126 0
```

Note that any minor will work, as this driver is not using the minor. Similarly, any name can be used because the only thing that identifies the driver in the kernel is the major number.

number.

7.2 smallchar-udev.ko

The next example is a more "modern" version of *smallchar*. It registers a *class* in *sysfs* and then creates one device, with minor number 0 (but any number will work). If you are running *udev* or another *hotplug* system, the device will automatically appear in your `/dev` directory. The name chosen is `/dev/chardev`; thus, all strings used in the source to name things are different, and you can check which of the strings is used in the various contexts.

This is how normal char drivers nowadays register their own entry points, in order for them to automatically appear in the filesystem.

The driver introduces the following concepts:

- Creation of a class of devices.
- Creation of a device within the class.
- Use of `IS_ERR` and `PTR_ERR` from `<linux/err.h>`.
- Use of `goto` for error management.

Note that there is another interface for automatic device creation, which may be more useful if you driver just needs to create an array of similarly-named devices (like `/dev/hw0`, `/dev/hw1` and so on). The function *alloc_chrdev_region* allocates and registers a range of minor numbers and returns the associate major number (dynamically allocated), while *register_chrdev_region* registers a range of minor numbers from a major number you already own. In both cases the *hotplug* mechanism is notified.

7.3 smallmisc.ko

The last char driver example is *smallmisc*. This is a char device like the other two (a 64-byte buffer, same semantics), but it is registered as a *misc* device. This means the major number is always 10 (reserved for *misc* devices) and you get a single minor.

The minor being used here is `MISC_DYNAMIC_MINOR`, so you don't even need to choose your own unique minor number. The device will automatically appear in `/dev` thanks to the *hotplug* system, but if you need to *mknod* by hand, you can find the minor in `/proc/misc`, using *grep* or similar tools.

The trick put in action here is simple: when the special file is being opened, the *open* method of the misc major number gets called; it then scans the list of registered minors and replaces its own *file_operations* with the ones of the client module.

Thus, *smallmisc* is simpler than *smallchar-udev* while being *hotplug-aware* just the same. Note that unless you are creating only one (or a pair of) special files, this is not the preferred way to work, and you should rather register your class and devices by yourself.

The source code uses `sc_` as prefix for all functions and variable, instead of `sm_` as you may expect, in order to ease users of *diff* who want to check what is different here and `smallchar.c`.

8 Sleeping

Simple drivers like *smallchar* can be used for trivial tasks like lighting a led or reading some input channels, but for any real work more is needed. A process reading from a device may need to sleep while waiting for data; similarly, a process writing may need to sleep while waiting for buffer space to become available.

The next examples show the minimal sleeping mechanism and a more complete setup.

8.1 smallsleep.ko

The smallsleep module declares a wait queue in order to put a process to sleep. It registers as a *misc* driver with dynamic minor, so you won't need to *mknod* manually.

When a process reading from the associated device, it is put to sleep until some other process writes to the device. The reading process will then get `EOF`, so it can be tested by simply running `cat`. When you write to the device, the sleeping `cat` sees end-of-file and terminates.

The driver introduces the following concepts:

- Wait queues and awakening.
- *wait_event_interruptible* (which re-evaluates the second argument).

8.2 smallsleep2.ko

Whereas *smallsleep* has the basics of sleeping, real code should support the `poll` and `select` system calls whenever `read` or `write` may sleep. In this case, after the file is reported as readable, the *read* method cannot sleep; thus, the implementation of *read* must be different, to handle consistency with *poll*.

The behaviour of *read* is different from `smallsleep`: here one byte is returned (the letter `x`) each time `read` returns. This allows testing with programs like `userspace/selread`, that print to *stdout* what is read from the device file.

The module is more difficult than the previous one because of the need to keep private data within the file; to allocate and initialise such private data we also need an *open* and *release* method. The system ensures that *release* is called only once for every file, even if the *close* system call may be called more than one on the same file (this happens if the user used `fork` or `dup` on this file).

Finally, when a system call may put the process to sleep, it should support non-blocking operations, by returning `EAGAIN` when the call would block.

The driver, therefore, introduces the following concepts:

- Use of `private_data` in a file.
- Kmalloc and kfree.
- The *open* and *release* methods.
- Use of `O_NONBLOCK` to support `EAGAIN`.
- Use of the *poll* method.

9 Timers and Works

The kernel offers primitives to execute operations in the future. The most common tool here is the timer; while in some cases it is being replaced by other mechanism, it remains a very simple and effective tool.

Being run in interrupt context, the timer is sometimes not suitable to perform the needed task. To this aim work queues have been introduced; they run in the context of a process and code in the work queue can thus sleep.

9.1 smalltimer.ko

The module registers a timer that re-registers itself. The timer has a period of 1s by default, and runs forever. The timer function just prints the current time (in `jiffies`) and the time where it was registered to run, for comparison.

The module accepts two parameters: `periodms` (default is 1000) and `count` (default is 0 == 4G). Thus, you can run the timer every 1 ms (or even 0ms), by limiting the number of iterations you can avoid locking up the system in interrupt overload.

The timer receives an `unsigned long` argument, but in all Linux architectures this type has the same size of pointers: whenever you need a pointer argument instead of integer, thus, using a cast is idiomatic and now considered bad.

The argument value is declared in `setup_timer` but there is no official way to change it. For this reason the module uses an external integer variable to store the expire time, using the argument as a pointer to that value. You may object the the `expires` field could have been used, but there are to reasons to avoid that. First, direct access to structure fields is to be avoided, to hide the internals of data; then, recent kernels add some slack to the timer expiration time (up to 0.4% by default), and reading back the value will make all those slacks to integrate over time.

To summarise, the driver introduces the following concepts:

- Use of `setup_timer`.
- Use of a pointer as timer argument, despite the mandatory cast.
- Using `mod_timer` instead of `add_timer` to avoid direct access to internal fields of the structure.
- Use of `msecs_to_jiffies` for time conversions.

9.2 smallwork.ko

Timers run in atomic context, because they are fired off the timer interrupt. A work queue, on the other hand, runs in the context of a process. There a dedicated worker process for each CPU in the system.

This module registers a work and a delayed work: the work is run “immediately”, while the delayed work is run some time in the future, according to a delay specified in *jiffies*.

The scheduled function is the same for both works, and it prints the time elapsed since the time it was registered. The delayed work is scheduled for 100ms in the future. This is the expected result:

```
[305746.022264] sw_init: process insmod (8384)
[305746.022284] sw_run: process kworker/0:2 (769) -- delay 11
[305746.122203] sw_run: process kworker/0:2 (769) -- delay 99926
```

The driver introduces the following concepts:

- Use of an immediate work.
- Use of a delayed work.
- Accessing the name and pid of the current process.

10 Interrupts

The three modules in this chapter deal with interrupts. The modules piggy-back on another interrupt source, for example your network card or disk drive. The handler is registered as a *shared* one, so that when interrupt events arrive both the original handler and the one of the simple module are called.

Please note that if the original handler doesn't allow sharing, these modules will the `EBUSY` – for example when hooking on the timer interrupt, irq 0 on the PC.

All the modules require an `irq` integer argument when being loaded. If no argument is passed, a message is printed to *syslog* and you'll need to unload and reload the module.

10.1 smallirq.ko

This module is the minimal interrupt handler. It simply counts the events it receives. It prints the number of events it processed at most once per second.

For example, when loaded under *qemu* piggy-backing on the network card (argument: `irq=11`), I get this under flood ping:

```
[57978.000137] si_handler: irq 11: got 14982 events
[57979.001474] si_handler: irq 11: got 16098 events
[57980.000316] si_handler: irq 11: got 15930 events
```

The driver introduces the following concepts:

- Registering and releasing an interrupt.
- Using a `devid` pointer as irq data structure;
- Using *jiffies* to print on a timely basis.

10.2 smallirq-tlet.ko

The module uses a *tasklet* to perform most of the work related to interrupt handling. While the previous module just counted the interrupts, this one counts both the interrupts and the tasklet invocations, reporting the delay between the interrupt and the tasklet.

The time delay is measured by saving the time stamp of the interrupt in the data structure, and then taking another stamping in the tasklet.

The driver needs some locking, because there are two critical sections. First, the `struct timespec` is not updated atomically, so saving the stamp in the data structure must be protected; then, `irq_count` is subject to read-modify-write operations in both the interrupt handler (it increments the count) and the tasklet (it reads and zeroes the counter).

A spinlock over the data structure is used to deal with both problems. Note that the two contexts (interrupt handler and tasklet) use

This is, for example, the output I get on a physical computer:

```
[4905606.305803] si_tlet_fun: process emacs (3713)
[4905606.310377] si_tlet_fun: irq 18: got 39 irq, 39 tlet
[4905606.315607] si_tlet_fun: delay ns: min 1876, avg 427531, max 16338531
```

It's apparent that the tasklet runs in the context of another process (here is *emacs*, but it may be any process or the idle task). Also, sometimes (under heavy load) the number of tasklets being run is less than the number of interrupts.

The driver introduces the following concepts:

- Registering a tasklet.
- Using *getnstimeofday*.
- Trivial use of spinlocks.

10.3 smallirq-work.ko

This last module performs the same functionality of `smallirq-tlet` but uses a *work* structure. The difference is that while a tasklet runs in atomic context (so called, *soft-interrupt*), the work runs in the context of a process. Besides replacing the tasklet with the work queue, there a few more details being addressed here.

First, note that interrupts are enabled, both when tasklets and works are running. Therefore, the previous module, has a minor race condition still pending here:

```
getnstimeofday(&ts);
spin_lock_irq(&d->lock);
nanodiff = (ts.tv_sec - d->irq_time.tv_sec) * NSEC_PER_SEC +
        ts.tv_nsec - d->irq_time.tv_nsec;
spin_unlock_irq(&d->lock);
```

The problem in the code above is that and interrupt can still happen after *getnstimeofday* and before the spin lock is taken. Thus, the `nanodiff` above may be negative. This module fixes the issue by running *getnstimeofday* after taking the lock.

Another issue is that the default `schedule_work` function activates the work in a reentrant context, and you can face errors like this:

```
[ 2219.173118] si_work: process events/0 (7)
[ 2219.173258] si_work: process events/1 (8)
[ 2219.173262] si_work: irq 18: got 1 irq, 2 works
[ 2219.173265] si_work: delay ns: min 0, avg 16664, max 23955
[ 2219.191896] si_work: irq 18: got 1 irq, 0 works
[ 2219.196713] divide error: 0000 [#1] PREEMPT SMP
```

What happened here is that during an interrupt burst two processors started executing the work: the first zeroed the `d->work_count` field and the other then calculated the average over 0 items.

This may be worked around by locking the whole work function, but a better solution is scheduling the work on the non-reentrant work queue:

```
queue_work(system_nrt_wq, &d->work);
```

Unfortunately, the non-reentrant queue only exists since 2.6.36, so a different approach is needed for 2.6.35 and earlier. Rather than spinlocking the whole work (which can be pretty long because of the *printk*, especially if you have a serial console like I do), the suggested code locks only the data collection and the decision about whether or not to print. Another option may be using semaphores, but I'd better not prevent execurion of other works just because ours is busy with *printk*.

This is the result on a dual-core system:

```
[20086.000289] si_work: process events/0 (7)
[20086.004489] si_work: irq 18: got 6964 irq, 6949 works
[20086.009746] si_work: delay ns: min 1158, avg 9655, max 109950
[20087.000311] si_work: process events/1 (8)
[20087.004514] si_work: irq 18: got 6734 irq, 6717 works
[20087.009659] si_work: delay ns: min 1237, avg 9694, max 54061
```

As you see, the work runs on either CPU and our locking is properly working in a non-intrusive way – the verification using the numbers shown above is left as an exercise to the reader. Also, I'm well aware that de delays being reported are not meaningful in all cases. Whenever you see more interrupts than executions of the work queue, that the delay is underestimated: another interrupt occurred after the work was scheduled, and this is the delay that was measured.

As a final remark, please note that it's very unlikely for an interrupt handler to request execution of a work in process context in a concurrent way.

11 Printing

The next two modules are related with diagnostic messages. The former is used to turn the application logs into the *printk* stream, and the latter shows how to declare your own console channel.

11.1 devprintk.ko

The *devprintk* module, though simple, is a real tool that I have been using in a few embedded projects. It is a misc device that sends to *printk* all the data that anybody writes to `/dev/printk`.

If your system is running unattended, like most control systems do, you can redirect *stdout* and/or *stderr* of the application to `/dev/printk`, to have a single messaging channel (the *printk* infrastructure) for your whole system.

The module adds the process name and `pid` to all messages it prints, so you can identify the individual messages in the overall stream; applications are expected to call *write* with one complete text line each time, otherwise the message stream won't be very friendly.

To create the string being passed to *printk*, the drivers uses a global buffer. In a multi-processor system this creates a race condition; the chosen solution here is using a *mutex*, a binary semaphore that puts the current process to sleep until the resource si free. Note that a *mutex* can only be used for code in process context, but this is the perfect situation for it.

In `userspace` you find `udplogs.c`. The program opens `/proc/kmsg` and sends to UPD broadcast every message it picks up from there. The trivial tool allows to send out the diagnostic stream to any observer that may plug a device on the network. While not suitable for internet-connected sites, control systems may well send out their diagnostic messages in the local network of the control site. By the way, this is exactly what high-level car and truck engines do: they are connected to a CAN bus where all status information of the engine is sent as broadcast; the pitcock, then, is just a sniffer that turns such information into leds and arrows and speed meters.

The *devprintk* module and *udplogs* show the following concepts:

- Registering a write-only device.
- Using a mutex for your critical section.
- Use of `/proc/kmsg`.
- Use of UDP broadcast.

11.2 smallconsole.ko

The module registers a *console* data structure, that will receive all *printk* traffic together with all the other consoles in the system. This is how you all diagnostic channels in Linux work, whether they are the VGA screen, the serial port, or the line printer.

The console is registered with `small` as its name and `CON_ENABLED` as flags – otherwise the console would only be enabled if a `console=small0` would be passed on the kernel command line.

The output it receives is fed back to *printk*, by only reporting how may bytes the new console received. To avoid recursion any message that includes the string `smallcons` is discarded.

This is what happens when you load the module:

```
[135434.016434] console [small-1] enabled
[135434.017415] smallcons: got  42 bytes
[135434.018419] new console loaded
[135434.019415] smallcons: got  35 bytes
```

The first message is generated by the kernel and sent to the new console as well, which counts it as 42 bytes of information. The next message is printed by *smallcons* itself, in its own init method, and again it gets fed to the new console, which reports it as being 35 bytes long.

Table of Contents