Linux Kernel Driver Tutorial

Martin Waitz
Friedrich-Alexander-University of Erlangen-Nuremberg, Informatik 3

tali@admingilde.org

1. Introduction

This tutorial was written for a course in Hardware and Linux driver development (DHWKLINUX (http://www3.informatik.uni-erlangen.de/Lehre/DHwkLINUX/WS2004/index.html)) at the Friedrich-Alexander-University of Erlangen-Nuremberg (http://www3.informatik.uni-erlangen.de). I hope it is interesting for other Kernel developers, too.

And well, sorry for the parts that are still unfinished...

Wozu braucht man ein Betriebssystem? Koordinierung von verschiedenen Programmen. Abstrahierung und Virtualisierung von Hardware.

BS bieten meist zwei unterschiedliche Schnittstellen an: eine zur Hardware und eine zu User-space Software Kernel reagiert immer nur auf Events, die jeweils aus dem Userspace oder von der Hardware kommen.

Walkthrough

arch/i386/kernel/entry.S

  syscalltable, irqs

kernel/irq/handle.c

  __do_IRQ, handle_IRQ_event

fs/read_write.c

  sys_read, vfs_read

2. Allgemeines

Fuer uns wichtige Punkte im Source Baum:

- Documantation/
- Documentation/CodingStyle
- include/linux -- API header files
- include/asm -- Symlink auf Architektur-spezifische Header
- kernel/ -- "Grundgeruest"
- arch/ -- Architektur-abhaengiger Code
- drivers/char/ -- Character-device-drivers
- fs/*.c -- Filehandling

sonstiges:

- init/ -- zum Booten des Kerns
- ipc/ -- Inter Process Communication (Semaphoren, shared memory, ...)
- drivers/ -- Treiber fuer alles moegliche (net, ide, input, pci, scsi, video,..)
- fs/*/ -- Dateisysteme
- mm/ -- Memory Management, alles rund um Pages
- net/ -- Netzwerk Code
- scripts/ -- Skrite die zum Bauen des Kerns benoetigt werden
- security/ -- Security Module, diese haben Veto-Rechte auf einzelne Operationen
- sound/ -- Sound-Treiber
- usr/ -- userspace code fuer initramfs, derzeit nicht verwendet

Debugging schwierig, also aufpassen! Kein floating Point SMP & kernel preemption muessen bedacht werden

Wichtigste Optimierung: Code sollte schnell zu verstehen sein.

Es gibt einige Kernel Debugging Optionen, die einem helfen ein paar haeufige Fehler zu entdecken.

2.1. Error checking

`#include <linux/err.h>` **`#include <linux/errno.h>`**

Check the return value! Most functions use negative return values to indicate an error. The actual value determines the kind of error (as defined in `include/linux/errno.h`). When an invalid parameter was encountered, -EINVAL is returned for example.

There are two different conventions for functions that return pointers:

1. return NULL in case of error (e.g. `kmalloc`)
2. return a special pointer indicating the actual error.

Be careful to check for the right condition! There are some macros to work with the second method:

IS_ERR(ptr)

return true if the given pointer is considered an error code

PTR_ERR(ptr)

return the (negative) error code represented by the pointer

ERR_PTR(error)

wrap the (negative) error code inside a pointer

If your function has to error-out, be careful to undo every step that already succeeded. The following structure is often used in the kernel to do that:

```
 ret = register_foo(...);
 if (ret < 0) goto out;

 ret = register_bar(...);
 if (ret < 0) goto out_foo;

 ret = -ENOMEM;
 obj = kmalloc(...);
 if (obj == NULL) goto out_bar;

 more_work();
 err = 0;
 goto out;

out_bar:
 unregister_bar(...);

out_foo:
 unregister_foo(...);

out:
 return ret;
```

When any subfunction returns with an error, we undo everything in the reverse order and pass the return code to the calling function, which will do similiar things on its own. This is a little bit like C++ exceptions.

2.2. Webseiten

- http://www.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html
- http://www.tldp.org/LDP/tlk/tlk-toc.html
- http://lwn.net/Kernel/
- http://kernelnewbies.org/documents/

2.3. Tipps

- Alle Funktionen und Variablen static machen.
- Alle Funktionen mit dem Treibernamen prefixen.

Dadurch gibts keine Konflikte mit vordefinierten Routinen und man sieht in einem Backtrace gleich wer schuld ist.

- Addressraum annotierungen: __USER, __iomem

- Anzeigen von aufgerufenen Systemcalls: `strace`

3. Adding new files to the build infrastructure

3.1. Configuration

The Linux Kernel can be built in many different ways. Most drivers can be included unconditionally, compiled as modules to be loaded at runtime or be left out completely.

The kernel configuration is stored in the file `.config` which can be created by calling `make menuconfig`. In order to create the menu structure that is presented by `make menuconfig` there a several `Kconfig` files in the source tree.

To add a new config option, edit the corresponding Kconfig file (for example `driver/char/Kconfig`) and add your entry:

```
 config EXAMPLE
  tristate "Example driver"
  depends on EXPERIMENTAL
default n
  help
    Say yes if you want to compile this example driver...
```

Full documentation for Kconfig files is located in `Documentation/kbuild/kconfig-language.txt`.

3.2. Building

Another step is needed to actually get the driver built into the kernel. Every directory contains a file called `Makefile` which gets included by the top level make file. There are two important variables that get initialized by all those make file snippets: `obj-y` and `obj-m`. `obj-y` holds all object files that get compiled into the kernel, `obj-m` holds all object files that get build as modules. For each configure option from the `Kconfig` file there is a variable that can be used in the make files. It evaluates to `y` if the option is to be include, `m` if the driver has to be build as module and `n` if it should not be compiled. The following construct is used very often to automatically do the right thing at compile time:

```
obj-$(CONFIG_EXAMPLE) += example.o
```

Full documentation of Makefile syntax is located in `Documentation/kbuild/makefiles.txt`.

3.3. Building out-of-tree modules

Sometimes it is too complicated to modify the kernel sources in order to include a new driver. The Linux build infrastructure has special support for out-of-tree modules. Everything that is needed to build a module for some kernel is installed to /lib/modules/$version/build, just next to all the kernel modules. You can use the following Makefile to have your module build by the regular kernel build infrastructure:

```
# build modules
obj-m = example.o

# use the kernel build system
KERNEL_VERSION := $(uname -r)
KERNEL_SOURCE := /lib/modules/$(KERNEL_VERSION)/build
all:
 make -C $(KERNEL_SOURCE) M=‘pwd‘ modules
clean:
 make -C $(KERNEL_SOURCE) M=‘pwd‘ clean
```

3.4. The Hello World module

`#include <linux/module.h>`

Kernel modules contain code that can be loaded into the running kernel, much like shared libraries in user space programs. They are located in `.ko` files in `/lib/modules/$version/kernel`. There are several utilities to work with them (included in the module-init-tools package):

lsmod

Print the list of currently loaded kernel modules.

modinfo

Display informations about the module.

insmod

Load a kernel module by specifying the full path name.

modprobe

Load a kernel module by module name. It is not neccessary to specify the full file name, it will be searched for in `/lib/modules`. If this module depends on other modules to work, then those will be loaded automatically, too.

rmmod

Remove a module from the running kernel.

module_{init,exit,param} MODULE_{LICENSE,AUTHOR,DESCRIPTION}

Ein Modul darf erst wieder aus dem Kern entfernt werden wenn es garantiert nicht mehr verwendet wird. Viele Datenstrukturen enthalten ein .owner Feld, das mit dem Makro THIS_MODULE initialisiert werden muss. Solange eine solche Datenstruktur in Verwendung ist, kann das dazugehoerige Modul nicht entfernt werden.

Funktionen die als __init gekennzeichnet sind werden nach der Initialisierung wieder aus dem Speicher entfernt. Funktionen die als __exit gekennzeichnet sind werden nur bei Modulen mitkompiliert.

4. Basic Kernel Infrastructure

4.1. Printing Messages

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

```
 printk(KERN_* "formatstring", ...)
```

aehnlich zu printf()

vor dem eigentlichen String kommt ein Prioritaetslevel:

```
 KERN_ERR, KERN_WARNING, ..., KERN_DEBUG
```

```
pr_debug("formatstring", ...)
```

Ausgabe mit KERN_DEBUG nur bei #define DEBUG

```
pr_info(...): Ausgabe mit KERN_INFO
```

Ausgaben sind praktisch immer moeglich, aber bei haeufig aufgerufenen Funktionen evtl. problematisch (z.B. Interrupt Handler)

Kritische Meldungen werden auf die Konsole ausgegeben. Die letzten Meldungen kann man sich auch per dmesg anzeigen lassen.

4.2. Allocating Memory

`#include <linux/slab.h>`

kmalloc & kfree -- kernel version von malloc & free

kmalloc hat im Gegensatz zu malloc noch ein Flags Argument: include/linux/gfp.h, meistens GFP_KERNEL oder GFP_ATOMIC. Nach Moeglichkeit GFP_KERNEL verwenden, aber geht nicht in kritischen Abschnitten oder Interrupt handlern, dort geht nur GFP_ATOMIC. GFP_ATOMIC findet allerdings haeufig keinen freien Speicher. Also: Speicherallokation moeglichst schon vorher mit GFP_KERNEL machen.

Auf jeden Fall: Rueckgabewert checken! (NULL bei Fehler)

Kontrolle ueber /proc/slabinfo

4.3. Accessing userspace memory

`#include <asm/uaccess.h>`

Userspace applications only have access to a limited

Sometimes the kernel has to access userspace memory. This memory may not be directly accessible even by the kernel, so there are some specialized functions to access that part of memory.

Access to userspace memory may fail if an invalid pointer was given. -EFAULT should be returned in this casse.

`access_ok(type, addr, size)`

Quick test to check for invalid addresses. *type* may be `VERIFY_READ` or `VERIFY_WRITE`.

`get_user(var, addr)`

read userspace memory at *addr* and store into variable *var*.

`put_user(var, addr)`

write variable *var* into userspace memory at *addr*

`copy_from_user(to, from, n)`

copy *n* bytes from userspace address *from* to kernel address *to*.

`copy_to_user(to, from, n)`

copy *n* bytes from kernel address *from* to userspace address *to*.

Be careful, the return values have different meanings for historical reasons: copy_to/from_user returns the number of bytes that could **not** be copied; the calling function is expected to return `-EFAULT` in this case. get/put_user already return `-EFAULT` or zero and access_ok returns 0 if the given address is invalid. You have to check the return value of get/put_user and copy_to/from_user even when access_ok says that the address is ok. The actual access to the memory could fail nevertheless.

4.4. Listing Objects

```
#include <linux/list.h>

 struct list_head {
  struct list_head *next, *prev;
 };
```

Liste haeufig:

```
   l -->  a  <--> b <--> _c
    \____________________/|
```

in Linux:

```
   l  <-->  a
   ^        ^
   |        |
   \/      \/
   c  <-->  b
```

Eine Liste ist immer eine zirkulaere Struktur, l->next->prev == l, _immer_ Eine solche Liste hat immer mindestens ein Element, damit sind die ->next und ->prev Zeiger nie NULL. Ein Element wird als "Aufhaenger" verwendet, die anderen zum Verknuepfen der eigentlichen Listen-elemente. der list_head besitzt keinen eigenen data pointer, sondern wird meistens direkt in die aufzulistenden Strukturen eingebettet.

```
INIT_LIST_HEAD(&list) creiert eine "leere" zyklische ein-element-Liste
list_add(&entry, &list) fuegt ein Element hinzu
list_del(&entry) loescht es wieder aus der Liste
list_del_init(&entry) wie list_del und INIT_LIST_HEAD zusammen
list_empty(&list) returns true if there is no other element in the list
list_entry(&entry, container_type, container_member)
list_for_each(iterator, &list)
list_for_each_entry(ptr, &list, container_member)
```

...

4.5. Error codes

EINVAL

Generic error with some parameter.

EINTR

The system call was interrupted by a signal.

ERESTARTSYS

If a system call is interupted by a signal and returns -ERESTARTSYS then it will be restarted from the beginning if the SA_RESTART flag is set in the siginfo_t structure that was used to register the signal Handler. The system call will return the error code EINTR if SA_RESTART was not set.

EAGAIN

The requested action is not possible at this time. The systemcall would have to wait for something but the userspace process requested nonblocking operation (e.g. with O_NONBLOCK).

EFAULT

attempt to access invalid memory.

EIO

communication to a hardware device failed or the hardware device indicated an error.

EBUSY

some essential resource is not available.

ENOMEM

out of memory

5. Files and stuff

Unix has one kind of abstract object that is used in many places: a file. All files have a common set of operations that are supported even when the actual implementation of these operations differ from file to file.

5.1. VFS overview

```
#include <linux/fs.h>
```

The virtual file system layer (VFS) is responsible to offer a unified view on files to user space programs. A file can be some storage object of a file system, a representation of a device driver, or a network socket. The actual file system, device driver or network protocol is responsible to implement all operations that can be called from userspace.

struct inode

represents a file

struct file

represents an open file, similar to a FILE* in userspace. The current position in the file is stored here.

struct file_operations

table of function pointers with implementation of the files behaviour

Such operations structures are very common to describe an abstract interface of some object. It is very much like the virtual method table of C++ objects.

Common operations on files are:

5.1.1. open(inode, file)

Called when a file is opened.

5.1.2. release(inode, file)

Called when the last filehandle to a file is closed.

5.1.3. read(file, buf, count, pos)

Called when data is to be read. *count* bytes should be transferred (via put_user or copy_to_user) to the userspace memory at address *buf*. *pos* holds a pointer to the current position inside the file. It should be updated to reflect the new position.

The function has to return the number of bytes actually copied (it is ok to return less then *count* bytes). If no data is available the function should wait for new data to arrive and then return that data. A return value of 0 indicates ’End of File’.

5.1.4. write(file, buf, count, pos)

Called when data is to be written. *count* bytes should be transferred (via get_user or copy_from_user) from the userspace memory at address *buf*. Same as the read function otherwise.

5.1.5. poll(file, polltable)

Called when a userspace process wants to wait for available data. Will be described later.

5.1.6. ioctl(inode, file, cmd, arg)

Called when ioctl(2) is called on this file. Ioctl provides a multiplexer for device-specific commands that do not map well on the standard read/write API. Example useages may be to set the transfer speed or to retrieve status information about the device.

5.1.7. file->private_data

Most operations only have the *file* pointer to identify the actual file that is needed. Device drivers usually maintain their own structure to describe each device. If a device driver supports more than one device then it has to map from the *file* argument its own device structure. For that reason does `struct file` contain the field `private_data`. This is an opaque `void*` pointer that can be freely used by the device driver. It is usually initialized by the open operation and then used by all other file operations.

5.2. Character devices

Device drivers can offer a file interface to talk to a device. Those files are usually located in `/dev`. Each file that is implemented by a device driver is identified by a device number (named `dev_t` inside the kernel) which is split into major and minor number for historical reasons. Usually the major number identifies the driver and the minor number identifies a device managed by this driver. `/dev` contains special inodes that include a major and minor number. The driver specific file operations are used whenever such an inode is opened.

There are two different types of device driver from the VFS point of view: character devices and block devices. Character devices operate on streams of bytes and usually do not support seeking while block devices only transfer fixed sized blocks. We only look at character devices here.

To create a new device node, see mknod(1).

The kernel maintains a list of currently available device drivers. A device driver can add itself to the list by calling `register_chrdev`. When doing so it must provide the major number which should be associated with the new driver and the file operations which should be called when a user space process access this device.

```
static struct file_operations example_fops = {
 .read = example_read, /* ... */
};
register_chrdrv(example_major, "example", &example_fops);
```

After the above call, `example_read` will be called whenever some process wants to read from a device node with the major number *example_major*.

The major number will be dynamically assigned when `register_chrdev` will be called with a `0` major_number. `register_chrdev` will return the actual major number in this case. Otherwise it will return 0 to indicate success. As usual, negative values indicate failure.

To remove the driver from the system:

```
unregister_chrdev(example_major, "name");
```

You can get a list of currently available drivers in `/proc/devices`.

6. Hardware access

6.1. ISA I/O

`#include <linux/ioport.h>` **`#include <asm/io.h>`**

Auf ein Geraet darf immer nur ein Treiber zugreifen. mit request_region(baseport, nr, "name") kann man sich den alleinigen Zugriff sichern (die naechsten request_region mit den gleichen Ports schlagen dann fehl). Mit release_region(baseport, nr) kann man das Geraet wieder freigeben. Kontrolle ueber /proc/ioports

lowlevel IO:

```
value = in{b,w,l}(port);
out{b,w,l}(value, port);
```

Many architectures support the concept of memory mapped IO. The control registers of hardware devices can be mapped into the normal address space directly. Access to such hardware is possible via direct assignments to the corresponding ’memory’ address.

To simplify hardware access for drivers, an IO abstraction has been included in newer Linux kernels. All Hardware IO Addresses can be stored in a `void __iomem *`. Access to the hardware is done via `io{read,write}{8,16,32}`, similiar to inb/out.

The kernel will automatically choose inb/outb or memmory mapped access based on the initialization of the `void __iomem *`.

For example:

```
iomem = ioport_map(port, nr);
ioport_unmap(iomem);
```

or

```
iomem = pci_iomap(...);
```

6.2. Interrupt handling

`#include <linux/interrupt.h>`

Device driver must be able to react on events that occur in the hardware. This is generally implemented via interrupts. If a device needs the attention of its driver, it generates an interrupt request (IRQ) which causes the CPU to enter the operating system at a defined address.

Usually there are several different interrupt lines that can be used to identify the device that caused such an interrupt. The operating system then uses the interrupt number to find the driver which is responsible for the device that generated an IRQ.

In Linux, a driver can ask the operating sytem to call a function each time an IRQ is generated:

```
 request_irq(nr, handler, flag, name, dev_id)
```

To remove the interrupt handler from the system, call `free_irq(nr, devid)`.

After calling `request_irq`, the function *handler* will be called each time an IRQ with number *nr* is generated. This handler will take the parameters *nr*, *dev_id*, and a structure holding the register contents of the processor at the time the IRQ was raised. The handler should return IRQ_HANDLED if it handled this IRQ or IRQ_NONE if the corresponding device was not responsible for this IRQ.

dev_id must be globally unique as it is used to identify the handler when freeing it, normally the address of a device data structure is used here.

name is only used for administrative purposes and is shown in `/proc/interrupts`.

flags is a combination of one or more of the following values:

`SA_INTERRUPT`

> all interrupts will be disabled while the handler is running. This should be used always.

`SA_SHIRQ`

> other drivers are allowed to register handlers for the same *nr*. That means that the handler may be called even when the corresponding hardware did not generate an interrupt and the handler function must be able to cope with unexpected interrupts. If possible, the driver should use SA_SHIRQ. That way it is possible to share interrupt lines with several devices. As interrupt lines are a rare resource in most machines, interrupt sharing has become a neccessity to support many devices.

`SA_SAMPLE_RANDOM`

> the timing of the IRQ is considered random and should be used to fill the entropy pool of the system.

Interrupt handlers have the highest priority in the entire system. The Linux kernel tries to execute them as quickly as possible, deferring all other things the CPU might be doing at that time. Other interrupts are usually disabled to always let one handler complete its operation even when there are many frequent interrupts. Interrupt handlers (and all other parts of the kernel that disable interrupts) should run as quickly as possible. Otherwise important interrupts could be lost or delayed.

To make interrupt handlers as fast as possible, they should only poll the hardware to look for the reason of the interrupt and should defer all lengthly operations. This will be described in the next section.

Be careful to enable interrupts in the hardware only after the kernel interrupt handler was successfully installed and to quiesce the device before removing the kernel interrupt handler.

6.3. Tasklets

`#include <linux/softirq.h>`

It is often neccessary to start lengthly actions like data transfers in response to a hardware interrupt. This is not allowed in a interrupt handler in order to keep overall interrupt latency low. Lengthly actions should be executed in a *Softirq* instead. Softirqs are started after normal interrupt processing (called hardirq) is finished and before control is given back to userspace applications. While softirqs are running, interrupts are enabled.

There are a number of predefined softirqs that run one after another. The most importat ones from a driver point of view are tasklets and timers. Tasklets allow to schedule one-shot actions that should be executed as soon as possible while timers allow to schedule actions that are executed later.

Tasklets are activated by `tasklet_schedule(tasklet)` or `tasklet_hi_schedule(tasklet)`. When scheduled by the `tasklet_hi_schedule` function, the tasklet will be run from the highest priority softirq, that is just after the normal interrupt handler returns. When scheduled by `tasklet_schedule`, it will be called from a low priority softirq, after the timer, network, and SCSI subsystems.

Tasklets have to be initialized before they can be scheduled. This is done by calling `tasklet_init(tasklet, handler, arg)`. When *tasklet* is scheduled, *handler* will be called with the sole argument *arg*.

7. Processes

`#include <linux/sched.h>`

All informations about one process is stored in `struct task_struct`. It includes the status, flags, priority, and many more information about one task. The task_struct of the currently running process is always available through the macro `current`.

Possible task statuses are:

TASK_RUNNING

> (**R**) The process is able to run and contributes to the system load. The scheduler decides which processes really receive CPU time.

TASK_UNINTERRUPTIBLE

(**D**) The process waits for some event. It will not be considered by the scheduler. The process cannot do anything while it is waiting (it cannot even be killed). This is usually used by device drivers while the process is waiting for some hardware to respond. Such a process contributes to the system load even though it will not receive CPU time; some other part of the system is considered to be working on behalf of the process.

TASK_INTERRUPTIBLE

(**S**) The process waits for some event as in TASK_UNINTERUPTIBLE but it can be woken up by a signal. This should be used when the action can be interrupted without side effects. A process in this state is considered asleep and does not contribute to the system load.

TASK_STOPPED

(**T**) The process is stopped by a signal (Ctrl-Z)

TASK_ZOMBIE

(**Z**) The process has exited but there are still some data structures around that could not yet be freed. The zombie will usually be freed when the parent calls wait4() to get the exit status.

7.1. work queues

FIXME

8. Waiting & locking

8.1. Spinlocks

```
#include <linux/spinlock.h>

 spinlock_t
 rwlock_t
```

Spinlocks are a way to synchronize access to data structures.

`spin_lock_init(lock)`

Initialize a lock

`spin_lock(lock)`

take a lock.

`spin_unlock(lock)`

release a lock.

If the lock is already taken by another processor, then `spin_lock` will simply try again until it succeeds. The processor does not execute anything else in this time. Therefor, critical sections guarded by a spinlock must be as short as possible.

When a datastructure is often read but only rarely updated then using a spinlock would be bad as only one reader could access the data at a given time. There is a special lock that takes into account such an asymmetric use.

`rwlock_init(rwlock)`

`read_{lock,unlock}(rwlock)`

Lock/unlock for reading; no writer can take the lock at the same time but many readers can hold the lock at the same time.

`write_{lock,unlock}(rwlock)`

Lock/unlock for writing; no other reader or writer is allowed to take the lock at the same time.

Spinlocks (and rwlocks) can be used almost everywhere, even in interrupt handlers. In order to prevent deadlocks, interrupts must be disabled during the critical section if the guarding lock may be used in an interrupt handler.

There is a special version of spinlocks that automatically disables interrupts on the local processor:

```
long flags;

spin_lock_irqsave(&lock, flags);
/* ... critical section, IRQs are disabled here */
spin_lock_irqrestore(&lock, flags);
```

8.2. Semaphores

`#include <asm/semaphore.h>`

```
struct semaphore;
```

Semaphores can be used to synchronize processes. A Semaphore is an integer variable which can be increased and decreased. When a process tries to decrese the value of the semaphore below zero, it is blocked until it is possible to decrease the semaphore without making it negative (i.e. until some other process increases the semaphore).

Unlike spinlocks, the blocked process is put to sleep instead of trying over and over again. As Interrupt handlers are not allowed to sleep it is not possible to use semaphores there. FIXME

`sema_init(sem, value)`

initialize semaphore.

`up(sem)`

increase semaphore (*V(sem)*)

`down(sem)`

decrease semaphore, wait if it would become negative (*P(sem)*)

`down_interruptible(sem)`

same as `down`, but returns -EINTR when it is interrupted by a signal.

A critical section can be implemented by initializing the semaphore to 1 and surrounding the critical section with down() and up() calls. It is advisable to use the interruptible version if the process can block for a long time.

```
ret = down_interruptible(&sem);
if (ret) goto out;
/* critical section */
up(&sem);
ret = 0;
out:
```

8.3. Wait queues

`#include <linux/wait.h>`

Wait queues are used to wake up a sleeping processes. A waitqueue is simply a linked list of processes that need to be woken up when some event occures.

`wait_queue_t`

an entry that represents one process that is to be woken up

`wait_queue_head_t`

linked list of `wait_queue_t` entries

`init_waitqueue_head(q)`

create an empty wait queue list

`wait_event(q, event)`

registers with wait queue *q*, waits until *event* becomes true.

`wait_event_interruptible(q, event)`

same as `wait_event`, but returns with -ERESTARTSYS when a signal is received.

`wake_up(q)`

wakes up all processes that are registered with *q*. If the process executed `wait_event`, then it will re-evaluate the *event* and eventually return from the `wait_event` function.

8.4. poll

`#include <linux/poll.h>`

The select/poll system call allows userspace applications to wait for data to arrive on one or more file descriptors.

As it is not known which file/driver will be the first to

The poll/select system call will call the `f_ops->poll` method of all file descriptors. Each ->poll method should return whether data is available or not.

If no file descriptor has any data available, then the poll/select call has to wait for data on those file descriptors. It has to know about all wait queues that could be used to signal new data.

`poll_wait(file, q, pt)`

register wait queue *q* for an poll/select system call. The driver should wake up that wait queue when new data is available.

This is best illustrated with an example. The following `example_poll` function returns the status of the file descriptor (is it possible to read or write) and registers two wait queues that can be used wake the poll/select call.

```
unsigned int example_poll(struct file * file, poll_table * pt)
{
 unsigned int mask = 0;

 if (data_avail_to_read) mask |= POLLIN | POLLRDNORM;
 if (data_avail_to_write) mask |= POLLOUT | POLLWRNORM;

 poll_wait(file, &read_queue, pt);
 poll_wait(file, &write_queue, pt);

 return mask;
}
```

Then, when data is available again the driver should call:

```
data_avail_to_read = 1;
wake_up(&read_queue);
```

This will cause the select/poll system call to wake up and to check all file descriptors again (by calling the `f_ops->poll` function). The select/poll call will return as soon as any file descriptor is available for read or write.

select/poll is often used by userspace applications to check if the next read or write of a file descriptor would block or not. However, bad things can happen between the select and the next read or write call. For example, another process could consume that data inbetween. If a process really expects a read or write call to not block, it should set `O_NONBLOCK` on open(2) or via fcntl(2). Every driver should check `file->f_flags` for `O_NONBLOCK` and return `-EAGAIN` if no data is available immediately.

8.5. Staying awake

There are some situations when sleeping is not allowed:

- in interrupt context -- without a process context, it is not possible to wake up later.
- while holding a spinlock.

This is important to keep in mind as many kernel functions may sleep for short periods of time internally. You have to make sure that you only call atomic functions while in interrupt context or while holding a spinlock.

Errors caused by calling the wrong function may be hard to find, as many internal functions only have a small chance of actually sleeping (e.g. kmalloc(..., GFP_KERNEL) only sleeps if it thinks that it can get more memory later). To make the user aware of a potential problem, `may_sleep()` can be used to print a warning if the function is executed in a context that does not allow to sleep.

9. atomic

`#include <asm/atomic.h>`

Manche Operationen brauchen kein locking um atomar zu sein: Zuweisungen zu Pointern und integern.

10. Timer

`#include <linux/delay.h>`

There are several functions that can be used to wait for some amount of time. They all start with one character that describes the unit of time that must be given as parameter: `s` for secons, `m` for milliseconds, `u` for microseconds, and `n` for nanoseconds.

`{m,n,u}delay(t)`

Spin the processor for the specified interval. These functions can even be used in interrupt handlers but they should not be used if the desired delay is greater than a few milliseconds.

`{s,m}sleep(t)`

Put the current process to sleep for the specified interval.

`#include <linux/timer.h>`

A timer can call a function at a specific point in time. Unit of time is 'jiffies', a counter that is monotonically increasing.

```
void my_timer(unsigned long data) {}

struct timer_list;
init_timer(&timer_list);
timer_list.function = my_timer;
timer_list.data = arg_for_my_timer;
timer_list.expires = jiffies + msecs_to_jiffies(msecs);
```

`add_timer(&timer_list)`

activate a timer

`del_timer(&timer_list)`

deactivate timer

`del_timer_sync(&timer_list)`

same as *del_timer*, but wait if a timer is still running

The timer is one-shot: it is automatically deactivated when the timer function gets executed. Both *del_timer* functions return 0 if the function was started and 1 if the timer is still active.

Be careful, the timer function is executed in interrupt context (by the softirq system). So it must not access the `current` pointer and must not sleep.

11. kfifo

```
#include <linux/kfifo.h>
```

Mit kfifo_put(kfifo, buffer, len) koennen Daten in das FIFO gespeichert werden, mit kfifo_get(kfifo, buffer, len) wieder entfernt werden. Der Fuellstand der FIFO kann mit kfifo_len abgefragt werden.

Initialisierung mit kfifo = kfifo_alloc(size, GFP_KERNEL, &lock); Wenn es immer nur einen Leser und einen Schreiber gibt wird kein locking benoetigt, dann kann man auch die __kfifo_* Funktionen nehmen.

12. Driver-Model

12.1. sysfs

A lot of information about the system is exported to userspace via the sysfs file system. It is usually mounted to /sys and contains the following sub-directories:

block

all block devices available in the system (hard drives, partitions, ...).

bus

bus types that are used to connect hardware devices. (pci, ide, usb, ...)

class

classes of device drivers that are available to the system. (net, sound, usb, ...)

devices

tree of devices connected to the system.

firmware

information gathered from the firmware of the system (ACPI)

module

list of currently loaded kernel modules.

Information is provided as one file per attribute. There are some standard attributes:

dev

the major- and minor-number of a block or character device. This can be used to automatically find or create the correct device node in /dev.

device

> a symlink to the devices directory. This can be used to find the hardware that is used to provide some service, for example the exact PCI device of the network card `eth0`.

driver

> a symlink to the drivers directory (located in /sys/bus/*/drivers)

A lot more attributes are available in /sys, depending on the bus and the driver used.

12.2. hotplug

Userspace programs can be started when the kernel finds new hardware devices. This can be configured via /proc/sys/kernel/hotplug. Usually a set of scripts is started that tries to find a driver which can handle the new hardware.

For example, each PCI driver can define a list of PCI vender/device IDs that it is able to handle. The device lists of all installed drivers are collected in the file /lib/modules/$kernelversion/modules.pcimap. The PCI controller calls the hotplug scripts when it finds a new PCI device. The correct driver for that hardware is looked up in modules.pcimap and loaded via `modprobe`. When the driver loads and detects more hardware (e.g. a disc controller enumerating all connected disk drives) then hotplug scripts may be invoked another time.

Using hotplug scripts, it is possible to autodetect most drivers that are needed in modern systems.

12.3. udev

The hotplug scripts are not limited to loading new drivers. There is a program that uses the information provided in /sys (in particular the `dev` attribute) to automatically create all the needed device nodes in /dev. `udev` hooks into the hotplug scripts to get notified when a new directory gets created in /sys.

The device node to create can be configured via some config files in /etc/udev. Please have a look at the manpages for details.

If you want to support automatic device node creation in your driver, you have to implement a `dev` attribute containing the major and minor number. This is described below.

12.4. Kernel infrastructure

- struct device
- struct bus

- struct driver
- struct class

12.5. How to export information from the driver

There are several possibilities to export information about your driver to userspace applications.

12.5.1. module parameters

All module parameters that are declared via `module_param` are automatically shown in /sys/module/$modulname/$paramname. The contents of the sysfs field always follows the current value of the correspondig module parameter variable. This is useful to obtain the value of some setting that may be autodetected by the module if no explicit parameter was given at module load time (e.g. for the major number or IO-base address that is to be used).

12.5.2. device/driver support by the bus subsystem

The kernel can detect a lot of settings automatically on most bus systems.

A PCI driver for example only has to fill in the contents of a `pci_driver` structure and all the rest (device creation, binding of devices and drivers, sysfs support) is automatically handled by the kernel.

12.5.3. class_simple

Making a driver show up correctly in sysfs using struct device/bus/driver/class can be very difficult if it is not already handled by the bus subsystem. Luckily, there is a small wrapper API that allows to publish driver information in /sys/class in a simple way: `class_simple`. It is mainly used to create a `dev` attribute which can be used by `udev` to automatically create device nodes for drivers.

class_simple_create(THIS_MODULE, "name")

> Creates a new driver class.

class_simple_destroy(class)

> removes that class.

class_simple_device_add(class, dev, device, "name%d", ...)

> Add a new device entry for this class. It will automatically get a `dev` attribute containing the major and minor number of the device. Those numbers are both stored in the *dev* parameter; use MKDEV(major, minor) to get it. *device* points to a `struct device` structure if one is available; it

is okay to use NULL here. The name of the new device is given with the last parameter(s); they contain one format string and an arbitrary number of format parameters, just like *printk*.

class_simple_device_remove(dev)

removes the device from the class, *dev* parameter must be the same as in device_add.

class_simple_create_file(device, attribute)

Add an extra attribute (besides the standard `dev`). *attribute* must be a pointer to a special variable which is created with the macro CLASS_DEVICE_ATTR(name, mode, show, store). This macro will create a variable `class_device_attr_`*name* representing the attribute *name*. It has the access mode as specified in *mode* (e.g. 0444 for read-only attributes). When the attribute is read or written then the *show* and *store* functions are called.

class_simple_remove_file(device, attribute)

remove an attribute from a device.