Linux kernel-space programming

Benedikt Waldvogel
mail at bwaldvogel.de

Unix Friends and User Group

November 30, 2006

Why kernel-space?

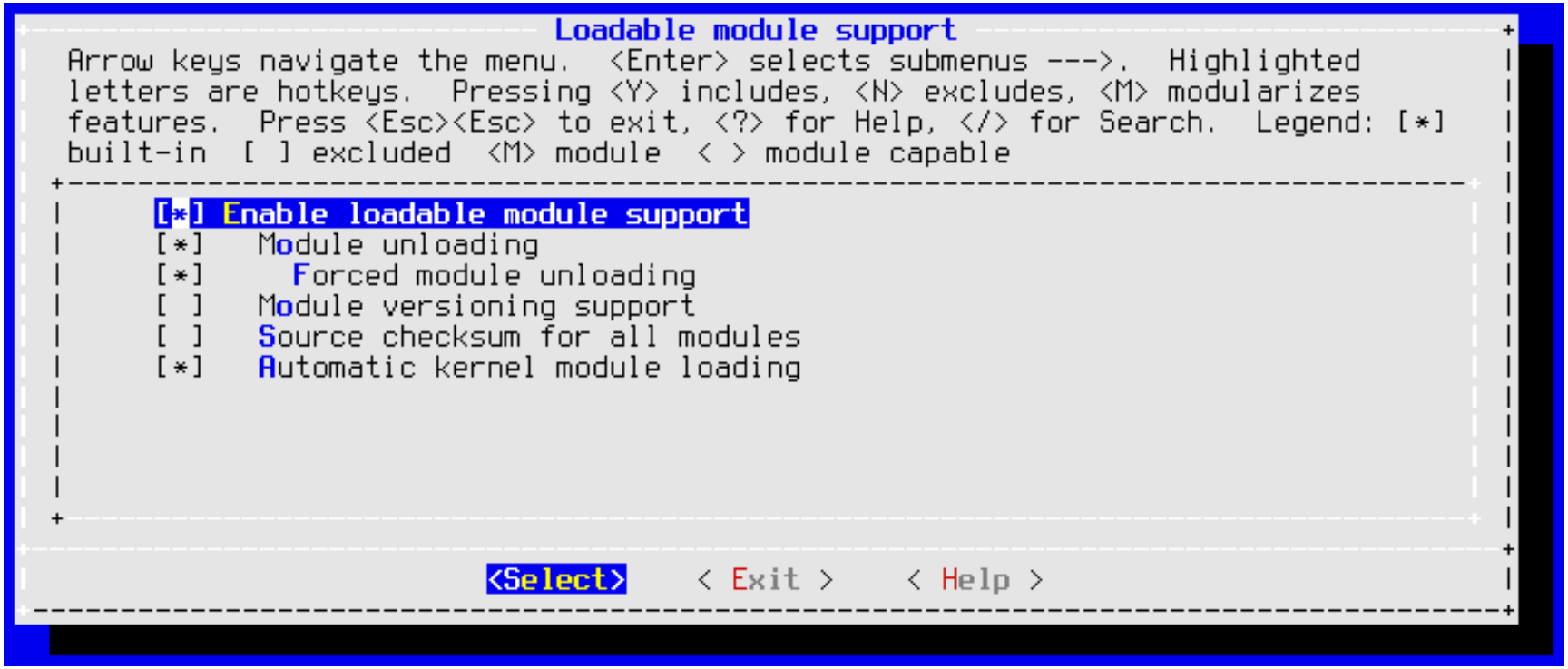
- *understand* how Linux works
- . . . or even *change* kernel behaviour (scheduling, networking)
- hardware access (interrupts, DMA)
- network/character/block devices
- root-kits ;-)

Prerequisites

- C know-how
- compiler collection
- virtual environment for testing (VMware, XEN)
- kernel headers and/or sources
- *recommended:* compile kernel by yourself (debugging options)

What is a module

- piece of code loaded/unloaded into the *running* kernel
- make sure `CONFIG_MODULES=y`
- modules are installed in `/lib/modules/`
- `# cat /proc/modules` to see which modules are loaded
- or `/sys/module/...`


Loadable module support
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted
letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*]
built-in [ ] excluded <M> module < > module capable
[*] Enable loadable module support
[*] Module unloading
[*] Forced module unloading
[ ] Module versioning support
[ ] Source checksum for all modules
[*] Automatic kernel module loading
<Select>
< Exit >
< Help >

How to insert a module

- `# modprobe module_name`
- resolves dependencies (`/lib/modules/2.6.xyz/modules.dep`)
- Example: `# modprobe msdos` becomes

```
insmod /lib/modules/.../fs/fat/fat.ko
insmod /lib/modules/.../fs/msdos/msdos.ko
```

Hello World

helloworld.c

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

int init_module(void)
{
    printk("Hello world!\n");
    return 0; /* success */
}

void cleanup_module(void)
{
    printk("Goodby world.\n");
}
```

the Makefile

```
obj-m += helloworld.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
	-rm *.ko
	-rm *.o
```

Metadata

- modules are usually enhanced with metadata
- you should at least provide the **license**

```
#include <linux/module.h>
MODULE_AUTHOR("Benedikt Waldvogel");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("unfug sample module");
...
```

Make, load and unload the module

```
# make && insmod helloworld.ko
# tail /var/log/messages
# rmmod helloworld
# modinfo helloworld.ko

filename:       helloworld.ko
author:         Benedikt Waldvogel
description:    unfug sample module
license:        GPL
vermagic:       2.6.18 mod_unload PENTIUMM REGPARM gcc-3.3
depends:
```

Define init/cleanup functions

```
static int __init module_init (void) { ... }
static void __exit module_exit (void) { ... }
module_init(module_init);
module_exit(module_exit);
```

if you forget the cleanup method, the module becomes **permanent** and can not be unloaded[1]

```
# lsmod
Module                  Size  Used by
helloworld               960  0 [permanent]
usb_storage            29636  1
...
usbcore                99456  usb_storage,uhci_hcd,ehci_hcd
```

[1] except if you force it (`rmmod --force`)

Use counter

```
void foo (void) {
    /* increase use counter */
    try_module_get(THIS_MODULE);
    ...
}
void bar (void) {
    ...
    /* decrease use counter */
    module_put(THIS_MODULE);
}
/* unload succeeds if use counter is 0 */
int cleanup_module (void)
{ ... }
```

typically `foo/bar` would be `device_open/device_close`

Coding style

> *First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.*

`/usr/src/linux/Documentation/CodingStyle`

Indentation, Brackets, Breaks

- 8-char indents, no spaces!
- one statement on a single line
- max length of lines: 80 columns
- **indent -kr -i8 helloworld.c**

```
int function(int x)
{
        if (a == b) {
                ...
        } else {
                ...
        }
}
```

Naming

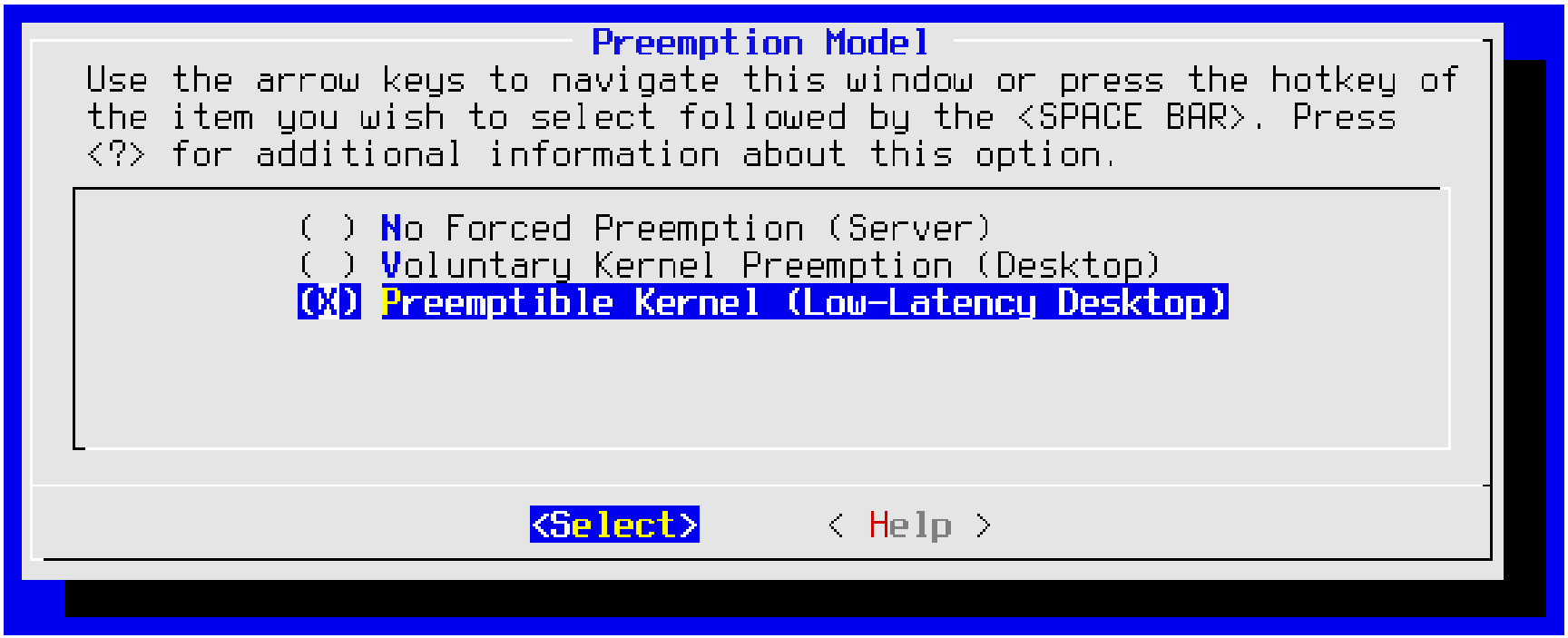
- descriptive **global** functions/vars
  (`count_active_users()` instead of `cntusr()`)
- short names for **local** variables
- don't use Hungarian notation
  (eg `ui_varname` for `unsigned int`)

Functions, Commenting

- up to ca. 50 lines, maximum
- don't inline functions with more than 3 lines
- max. 5-10 local variables
- explain **what** your code does, not **how** it's done
- all comments at function-head

Preemption

- kernel-space preemption possible since 2.6 (CONFIG_PREEMPT)



Dynamic memory

kmalloc/kfree	allocate up to 128k *physically continuous* memory
kcalloc(...)	same as kmalloc, but memory is zeroed
vmalloc/vfree	allocate memory in virtual address space (more than 128k possible but **no DMA**!)

void *__kmalloc__ (size_t *size*, int *flags*);

- flags declared in `<linux/mm.h>`

GFP_USER	associated userspace process sleeps until free memory available
GFP_KERNEL	associated kernel function sleeps until free memory available
GFP_ATOMIC	doesn't sleep (used in ISRs)

Timer ticks - Jiffies

The term jiffy (or jiffie) is used in different applications for various different short periods of time.
In general parlance, the term means any unspecified short period of time, or a moment, and is often used in the sense of the time taken to complete a task. The origin of the word is unknown, but it is believed to have first appeared in 1779.

`http://en.wikipedia.org/wiki/Jiffy`

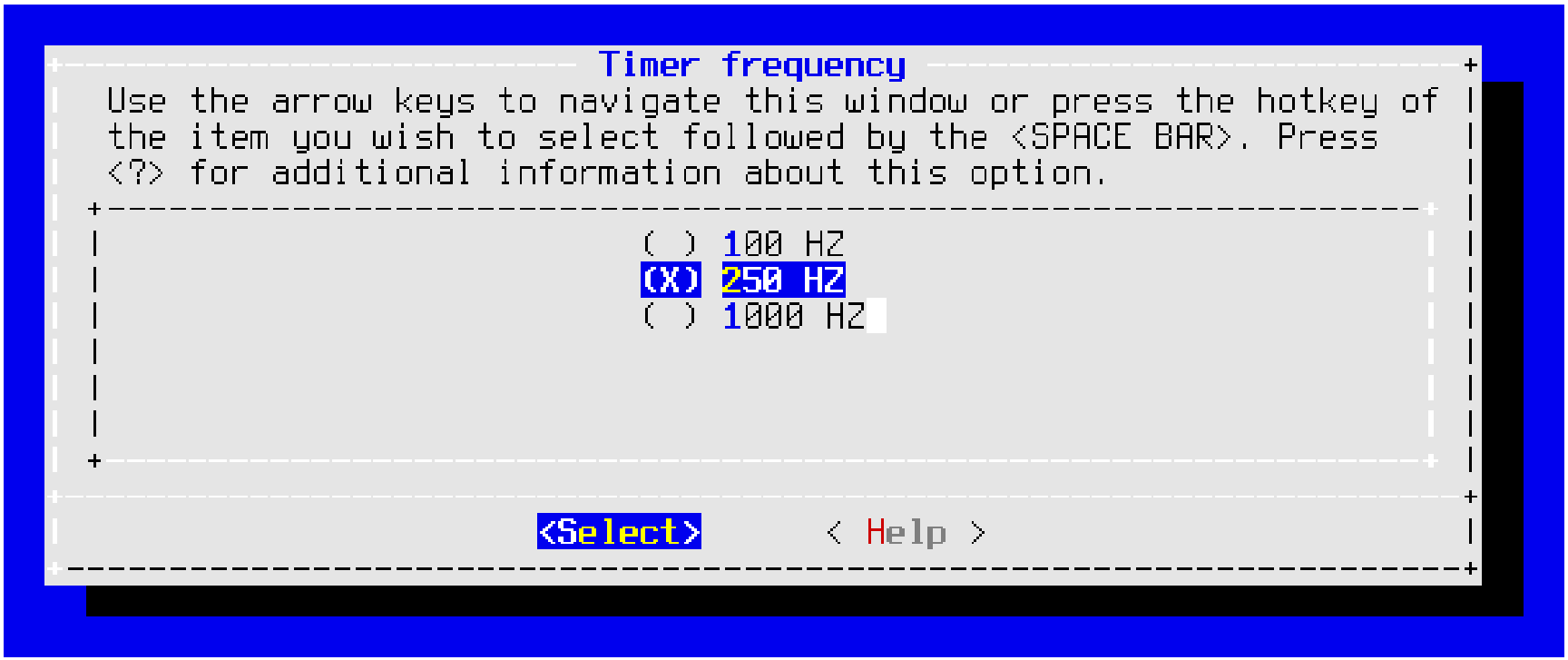
```
# cat /proc/jiffies && sleep 1s && cat /proc/jiffies
jiffies: 4025402
jiffies: 4025654
```

difference: 252

```
# cat /proc/jiffies && sleep 1s && cat /proc/jiffies
jiffies: 4034105
jiffies: 4034358
```

difference: 253
no coincidence! HZ + some overhead ...

HZ can be 100, 250 or 1000



older kernels used 100 HZ as default

current

- **current** is a macro which returns a `struct task_struct*`
- it points to the current process. Example:

```
printk("current process' pid: %d\n", current->pid)
```

/usr/include/linux/sched.h

```
struct task_struct {
    /* -1 unrunnable, 0 runnable, >0 stopped */
    volatile long state;
    struct thread_info *thread_info;
    int prio, static_prio;
    unsigned long long timestamp, last_ran;
    unsigned long long sched_time; /* time spent running */
    struct task_struct *parent; /* parent process */
    struct list_head children;
    struct list_head sibling;
    ...
}
```

Character device - how to provide /dev/mydev

1. `#include <linux/fs.h>`
2.
```
struct file_operations fops = {
    .read    = device_read,
    .write   = device_write,
    .open    = device_open,
    .release = device_release };
```
3. `register_chrdev(240, "mydev", &fops);`
4. `# mknod /dev/mydev c 240 0`
5.
```
# cat /proc/devices | grep mydev
240 mydev
```

Major numbers: `http://lanana.org/docs/device-list/devices.txt`

device_open, device_read examples

```
int device_open(struct inode *inode, struct file *file)
{
    try_module_get(THIS_MODULE);
    printk("device opened, use counter increased\n");
}

ssize_t driver_read(struct file *instanz, char *user,
                    size_t count, loff_t *offset)
{
    int not_copied, to_copy;
    char hello_world[] = "Hello world";

    to_copy = strlen(hello_world)+1;
    if( to_copy > count )
        to_copy = count;
    not_copied = copy_to_user(user,hello_world,to_copy);
    return to_copy-not_copied;
}
```

Major / Minor

Major associates a device with a certain kernel module/driver

Minor a driver is able to handle multiple devices with it

Example:

```
# ls /dev/my*
crw-r--r-- 1 root root 240, 0 26. Nov 11:45 /dev/mydev
crw-r--r-- 1 root root 240, 1 26. Nov 11:54 /dev/mydev2
```

- **mydev, mydev2** are both handled by driver which registered 240
- use Major numbers 240-254 for you own (local) devices

Network devices

- `register_netdev(...)`
- `unregister_netdev(...)`
- handle interrupts
- similar to character devices
- see `/usr/src/linux/drivers/net/isa-skeleton.c` which is a network driver outline

Procfs

- designed to export **process information** to userland

1. create /proc/example
   ```
   entry = create_proc_entry("example", 0644, 0)
   ```
2. assign read/write handlers:
   ```
   entry->read_proc = procfile_read;
   entry->write_proc = procfile_write;
   ```
3. int **procfile_read**(char *buffer, char **buffer_location, off_t offset, int buffer_length, int *eof, void *data) { ... }
4. int **procfile_write**(struct file *file, const char *buffer, unsigned long count, void *data) { ... }

Simple example

```
int procfile_read(char *buf, char **loc, off_t off,
                  int buf_len, int *eof, void *data)
{
    return scnprintf(buf, buf_len-1, "Hello world\n");
}

int module_init(void)
{
    struct proc_dir_entry *bl_proc =
            create_proc_entry("example", 0444, NULL);

    bl_proc->read_proc = procfile_read;
    bl_proc->write_proc = NULL; /* no writing possible */
    bl_proc->owner = THIS_MODULE;
    bl_proc->size = 40;

return 0;
}
```

Sysfs

- sysfs represents the kernel device model
- single key/value pairs instead of complex structures

kobject

- object management for your C code (eg. reference counting)
- exports **itself** to /sys
- an object is represented by a folder
- its attributes as files in that folder

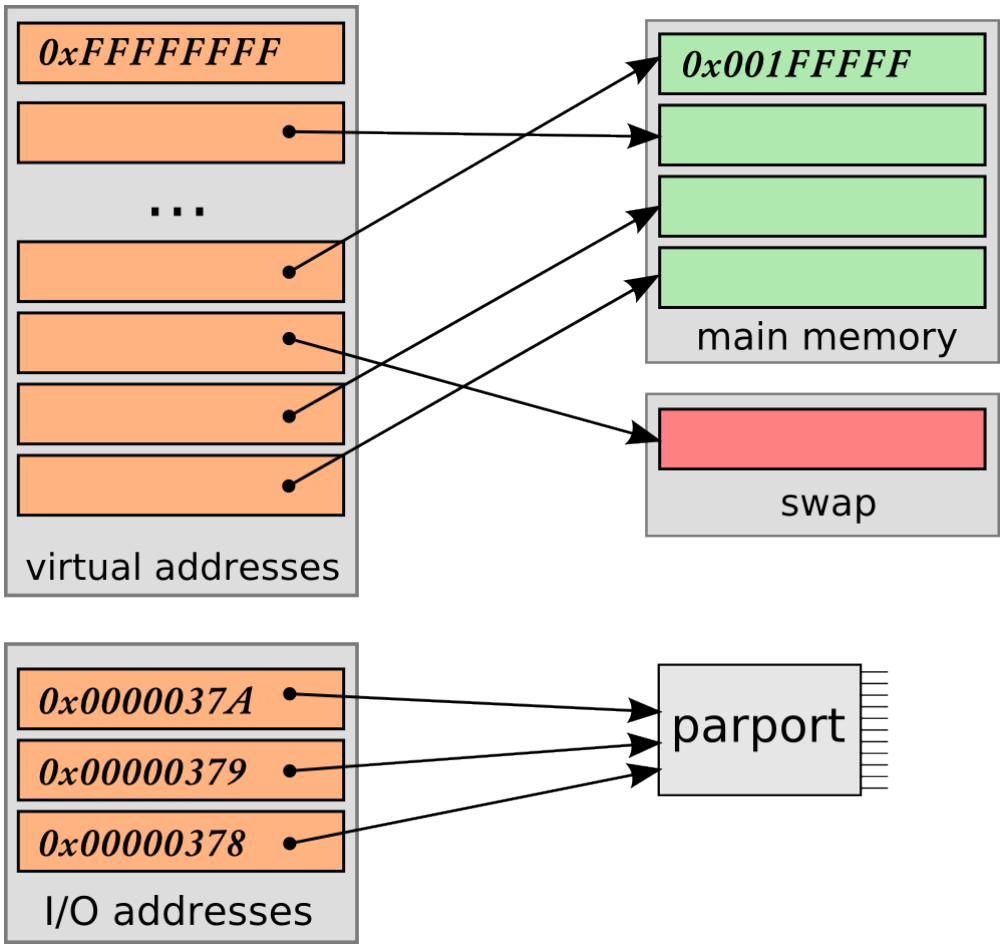
see /usr/src/linux/Documentation/kobject.txt

Module parameters

```
static int parm1 = 0;
module_param(parm1, int, 0644);
```

- either secifiy the parameter while insmodding:
  `# insmod helloworld.ko parm1=23`
- or use sysfs:
  `# cat /sys/module/helloworld/parameters/parm1`
- `0644` allows writing for root

Hardware access



Example: parallel port

- first of all we must register the I/O ports

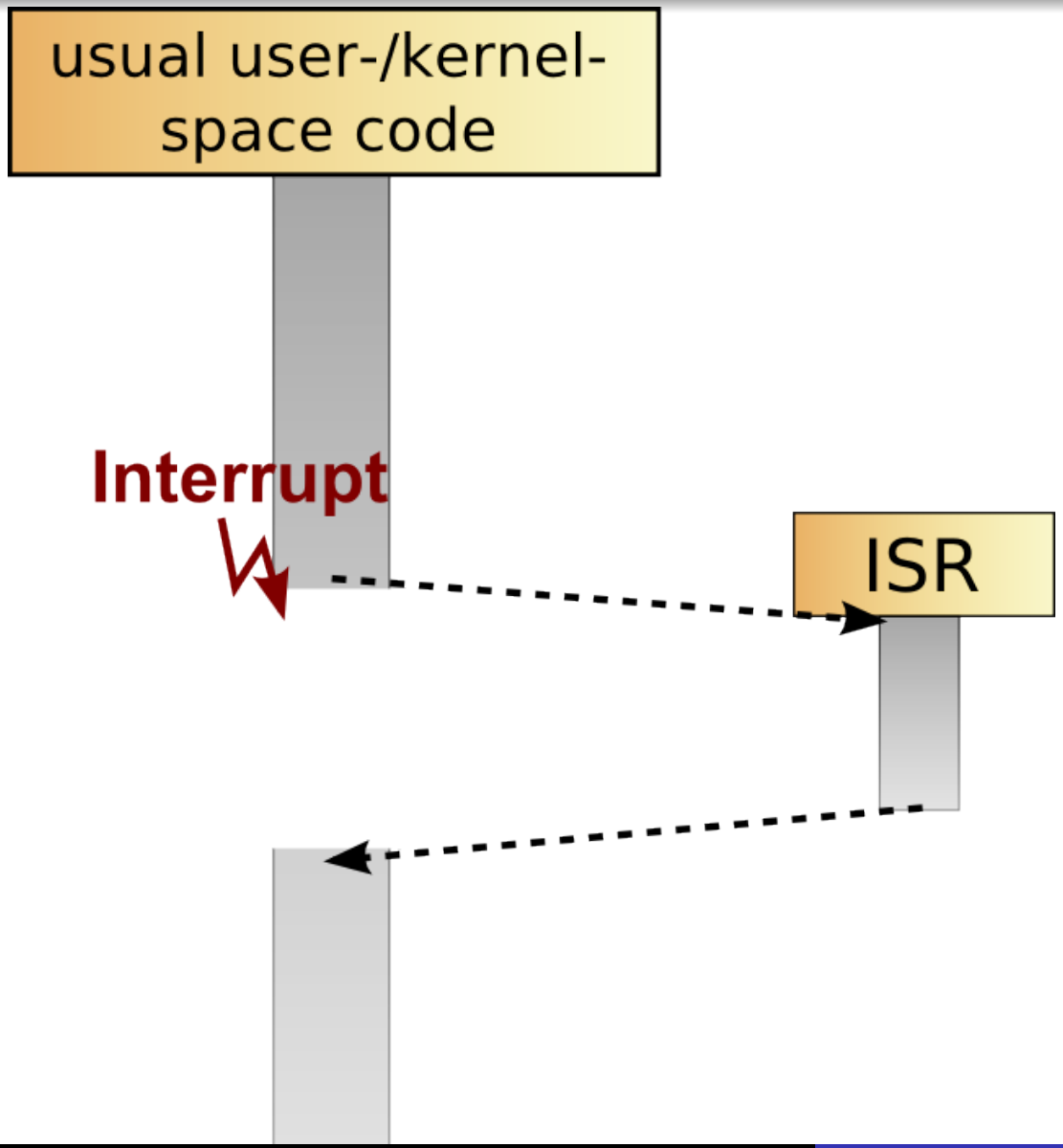
1. `#include <linux/ioport.h>`
2. `request_region(0x378, 3, "unfug");`
3. check `# cat /proc/ioports | grep unfug`
   `0378-037a : unfug`
4. don't forget to release the region:
   `release_region(0x378, 3);`

read/write data

- ...then we can read/write data to hardware registers
- `char *b = 0x378; *b = 0x23;`
- **not possible** on x86
- use special I/O commands instead
- `inb()`, `outb()`, `outw()`, `outl()` etc
- use `outb(0x378, 0x23);` to write the byte 0x23
- `char i = inb(0x378);`

When to read from Hardware?

- (active/intelligent) polling
- **Interrupt-driven**


usual user-/kernel-space code
Interrupt
ISR

Registration of an interrupt service routine (ISR)

1.
```
int isr(int irq, void *dev_id, struct pt_regs *regs)
{
    printk("interrupt on irq %d\n", irq);
    return 0;
}
```
2. `request_irq(7, isr, SA_INTERRUPT, "unfug", id);`
3. check `/proc/interrupts`

```
0:      138133    timer
7:           0    unfug
```

4. don't forget to free the irq: `free_irq(id);`

- note: no cpu regs parameter (`pt_regs`) since 2.6.19

Flags of `request_irq()`

- `request_irq()` takes a flags to set ISR behaviour

SA_SHIRQ	interrupt is shared between drivers
SA_INTERRUPT	deactivate other interrupts
SA_SAMPLE_RANDOM	use the interrupt to feed entropy pool

example:

```
request_irq(7, my_handler, SA_SHIRQ | SA_INTERRUPT | SA_SAMPLE_RANDOM, devid);
```

Interrupt Service Routines

Dilemma

- the ISR is possibly called *100 times* or more a second
- interrupts turned off during ISR [a]
- heavy disk access would interfere with network or clock

[a] if SA_INTERRUPT is set

Solution

- the ISR is divided in **upper halves** and **bottom halves**
- upper half is minimal, initiates a bottom half
- bottom halves (BH) do the complex part
- BHs in 2.4+ replaced by: **timers, tasklets, threads**

Timers

delay the execution of a certain function (*once*)

1. `#include <linux/timer.h>`
2. `static struct timer_list simple_timer;`
3. `init_timer(&simple_timer);`
4. `int timed_function(long data) { ... }`
5. `simple_timer.function = timed_function;`
6. delay the execution by 5 seconds
   `simple_timer.expires = jiffies + 5*HZ;`
7. `add_timer(&simple_timer);`
8. if module is unloaded before timer expired, delete it
   `del_timer(&simple_timer);`

Tasklets

queued execution - the tasklet is executed as soon as possible

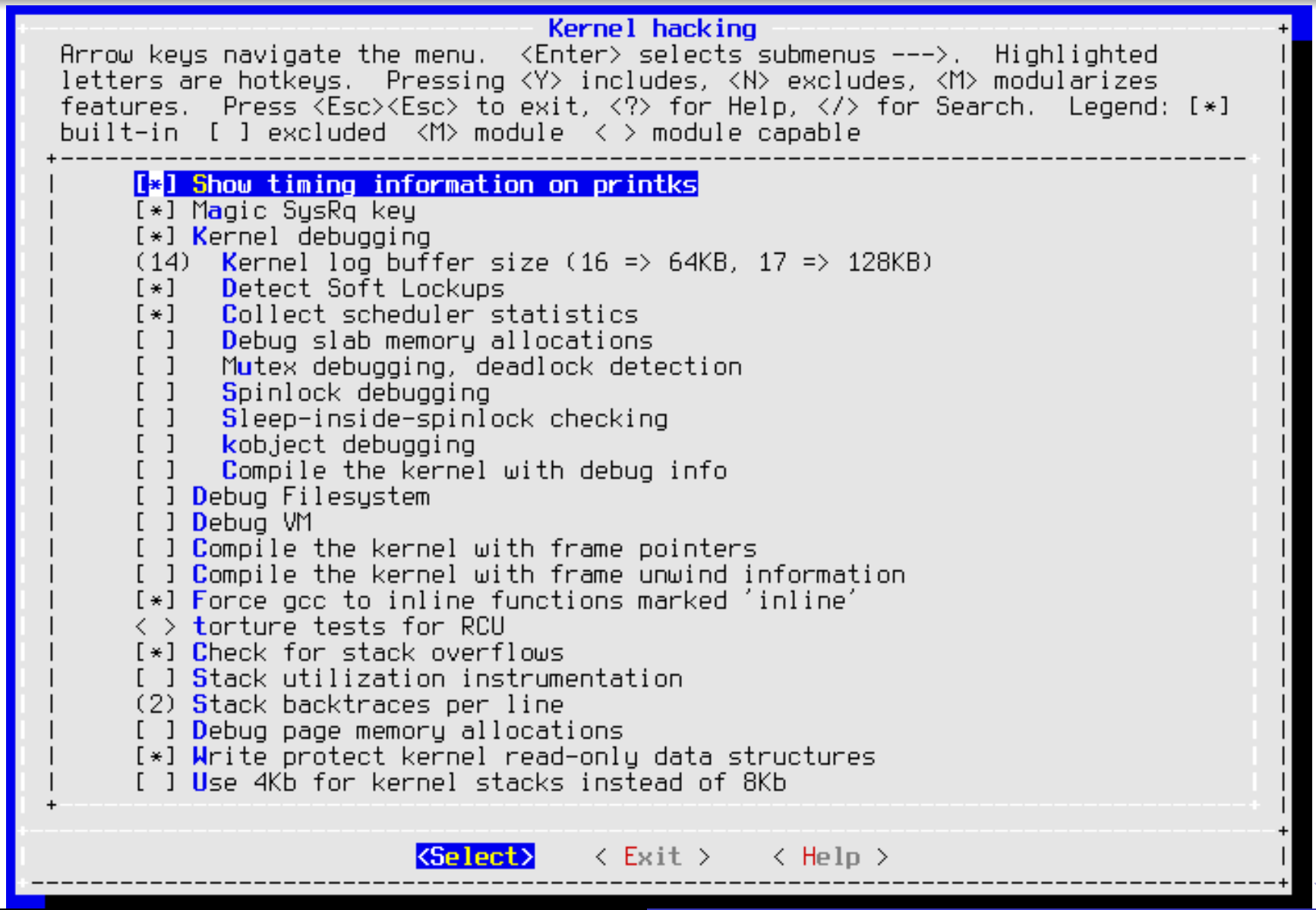
1. `#include <linux/interrupt.h>`
2. `void do_something (unsigned long data) { ... }`
3. `DECLARE_TASKLET(my_tasklet, do_something, data);`
4.
```
int interrupt_routine(...)
{
    /* do something fast */
    ...
    /* queue the more complex part */
    tasklet_schedule(&my_tasklet);
    return IRQ_HANDLED;
}
```

Threads

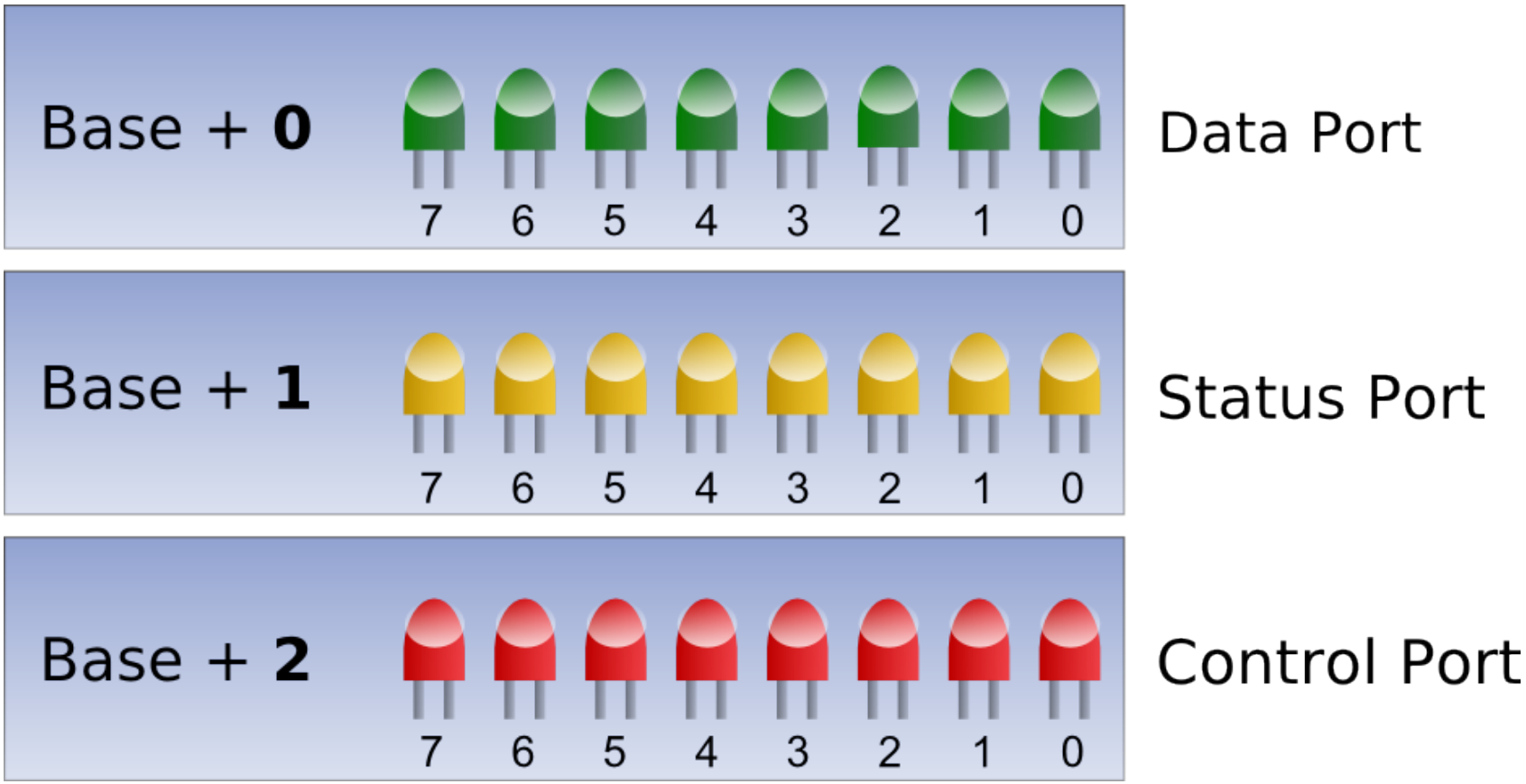
1. `#include <linux/sched.h>`
2.
```
static int thread_function(void *data)
{
    daemonize("unfug_thread");
    allow_signal(SIGTERM);
    /* do something or sleep until SIGTERM */
    ...
    return 0; /* exit thread */
}
```
3. `thread_id = kernel_thread(thread_function, 0, CLONE_KERNEL);`
4. kill the thread on module cleanup
   `kill_proc(thread_id, SIGTERM, 1)`

Output to syslog

- `#include <linux/kernel.h>`
- `printk(KERN_INFO "info text\n")`
- instead of `KERN_INFO` also possible (ordered by priority):
  `KERN_NOTICE`, `KERN_WARNING`, `KERN_ERR`, `KERN_CRIT`, `KERN_ALERT`
- `pr_debug("...");` output if kernel compiled with DEBUG
- `pr_info("...");`


Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable
[*] Show timing information on printks
[*] Magic SysRq key
[*] Kernel debugging
(14) Kernel log buffer size (16 => 64KB, 17 => 128KB)
[*] Detect Soft Lockups
[*] Collect scheduler statistics
[ ] Debug slab memory allocations
[ ] Mutex debugging, deadlock detection
[ ] Spinlock debugging
[ ] Sleep-inside-spinlock checking
[ ] kobject debugging
[ ] Compile the kernel with debug info
[ ] Debug Filesystem
[ ] Debug VM
[ ] Compile the kernel with frame pointers
[ ] Compile the kernel with frame unwind information
[*] Force gcc to inline functions marked 'inline'
< > torture tests for RCU
[*] Check for stack overflows
[ ] Stack utilization instrumentation
(2) Stack backtraces per line
[ ] Debug page memory allocations
[*] Write protect kernel read-only data structures
[ ] Use 4Kb for kernel stacks instead of 8Kb
<Select>
< Exit >
< Help >

Parallel port registers



Blinkenlights example

- `# insmod blinkenlights.ko speed=25`
- `/proc/blinkenlights`
- `/sys/modules/blinkenlights/parameters/speed`
- `/proc/ioports`
- `# insmod interrupt.ko`
- connect pin 10 with 22 and see `/var/log/messages`

the macros `likely()` and `unlikely()`

- you often read the macros `likely()` and `unlikely()`
- it's a compiler hint to optimize the branch

```
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
```

- Example:

```
void foo(int rand)
{
    if (unlikely(rand == 239482323L)) {
        ...
    }
}
```

export.c

```
void foo(void)
{
    pr_info("foo called\n");
}
EXPORT_SYMBOL(foo);
```

import.c

```
static int init_module(void)
{
    foo();
    ...
}
```

- see /**proc/kallsyms** for kernel symbols
- `# insmod export.ko`
- now you can insmod `import.ko`, which depends on `export`
- `# lsmod`

```
Module                  Size  Used by
import                  1280  0
export                  1536  1 import
```

```
#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module initialization */

static int skel_loader(struct module *m, int what, void *arg)
{
    switch (what) {
        case MOD_LOAD:                  /* kldload */
            printf("Skeleton KLD loaded.\n");
            break;
        case MOD_UNLOAD:
            printf("Skeleton KLD unloaded.\n");
            break;
        default:
            return EINVAL;
    }
    return 0;
}

/* Declare this module to the rest of the kernel */
static moduledata_t skel_mod = { "skel", skel_loader, NULL };

DECLARE_MODULE(skeleton, skel_mod, SI_SUB_KLD, SI_ORDER_ANY);
```

see http://www.captain.at/programming/freebsd/

Links

- `http://lxr.linux.no/` Cross-Referencing Linux
- `http://fxr.watson.org/` FreeBSD and Linux Kernel Cross-Reference
- `http://www.kernelnewbies.org`
- `http://www.lwn.net`
- `/usr/src/linux/Documentation/`
- Memory Management in Linux
  `http://www.cse.psu.edu/~anand/spring01/linux/memory.ppt`

Sources

- Linux Treiber entwickeln
  `http://ezs.kr.hsnr.de/TreiberBuch/html/`
- Linux-Geraetetreiber (kernel 2.4)
  `http://www.oreilly.de/german/freebooks/linuxdrive2ger/book1.html`
- `http://www.linux-magazin.de/Artikel/ausgabe/2004/05/094_kerntechnik10/kerntechnik10.html`
- The Linux Kernel Module Programming Guide
  `http://tldp.org/LDP/lkmpg/2.6/html/index.htm`

cleanup_module()

- Thank you for your attention, any questions?

get slides and examples from `http://www.bwaldvogel.de/kernelspace/`