Drivers in Linux: an Introduction

Kernel 2.6

Lecturer:
Ing. Luca Pizzamiglio
pizzamiglio@cefriel.it

Overview

- Introduction
- Devices
  - Different types of devices
  - Major and minor numbers
  - Static vs. Dynamic allocation
- Writing drivers
  - Initialization and cleanup
  - FOPS
  - Writing simple character driver

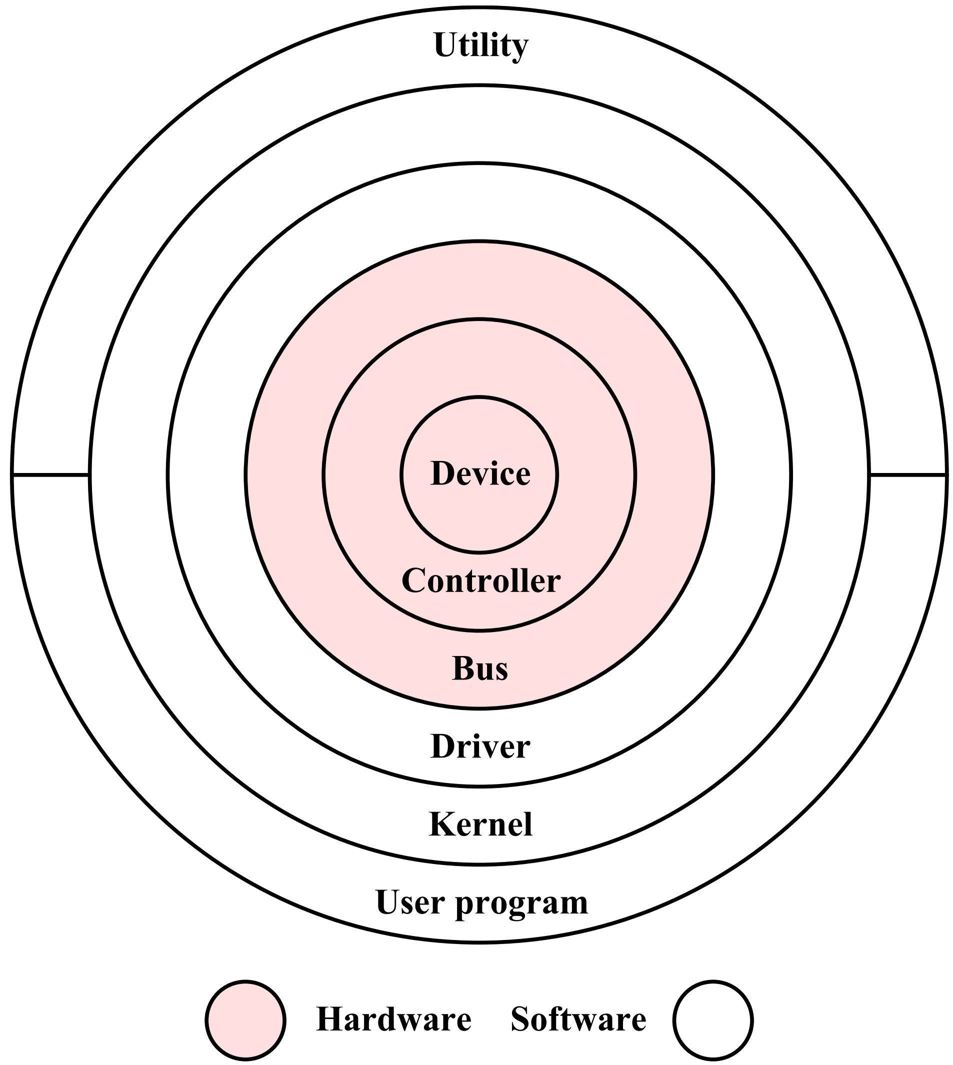
What is a Driver?

- An interface between the kernel and the hardware
  - Allows communication and exchange of data
  - Unifies the access to devices of the same type
  - Protects the hardware from wrong usage
- Various devices connected to the computer
  - Different kinds of devices, vendors, capabilities
- Physical devices are represented by files that can be accessed by standard syscalls
  - **`open`**, **`close`**, **`read`**, **`write`**

Second point of view

- A software layer between the application and the device
  - The driver programmer choose how the device should appear
    - Different drivers for different capabilities
  - The driver programmer has complete freedom to determine how to handle particular situations
    - Concurrency
- It makes the hardware available
- It leaves all the issues about how to use the hardware to the applications

Where a Driver Works



Problems Connected with Driver Writing

- There is no safety net between the driver and the computer
  - If a driver crashes, the system might too
  - There are some protections
- They are hard to debug
  - It means to debug a running kernel
  - There are specific tools
- Implementation problems
  - Concurrency using a single device by different programs

Policies Vs. Mechanisms

- Drivers should provide mechanisms
  - What capabilities should be provided
- Applications establish policies
  - How those capabilities can be used
- Trade-off
  - Time to write the driver vs. Driver's flexibility

- Example
  - A digital I/O driver offers only byte-wide access to a device to avoid writing the extra code to handle individual bits

Policy-free Advantages

- Different users have different needs!
- Flexibility comes if a driver exploits full capabilities of the hardware without adding constraints
  - Support for both synchronous and asynchronous operation
  - Ability to be opened multiple times
- Lack of software layers to “simplify things”
- Easy to debug and maintain
- Applications released with drivers
  - Help with configuration and access to the target device
  - Client libraries that provide capabilities that do not need to be implemented as part of the driver itself

Modes of operation

- Polling
  - It is wasteful in terms of processor time
- Interrupt
  - Must be supported by the hardware
  - The number of interrupts is limited
    - It is possible for different devices to share a single interrupt
      - If the hardware can be interrogated whether it generated an interrupt
      - If the ISR can forward an interrupt not triggered by the hardware it is controlling

Interrupts

- Slow and fast routines
  - Can(not) be interrupted during execution
- It is important to reduce to the minimum the lapse of time with interrupts disabled
- There are pieces of code that need to run without interrupts
  - Bottom halves
    - Pieces of code that do not need to be serviced immediately and can be interrupted
    - Their execution can be deferred later when interrupts have been enabled again
    - Functions that really need it can be run as fast interrupts
  - Task queues: an extension of the concept of b.h.

DMA

- Direct Memory Access
- Hardware must support it
- The kernel must include the support for
- It can be used to transfer data from and to the memory
  - It relieves the processor of the computational load of the control
  - It allows other task to be handled

Overview

- Introduction
- Devices
  - Different types of devices
  - Major and minor numbers
  - Static vs. Dynamic allocation
- Writing drivers
  - Initialization and cleanup
  - FOPS
  - Writing simple character driver

Devices in Linux

- Devices are usually reflected in the filesystem
  - `/dev` directory
- Examples
  - Hard drives
    - IDE : /dev/hda, /dev/hda1, /dev/hda2, …
    - SCSI : /dev/sda, /dev/sda1, /dev/sda2, …
  - Serial ports : /dev/ttyS0, /dev/ttyS1
  - Parallel ports : /dev/lp0, /dev/lp1
  - SCSI streamer
    - Rewinding : /dev/st0, /dev/st1
    - Non-rewinding : /dev/nst0, /dev/nst1

Device types

- Character devices
  - Serial and parallel ports, streamers
  - Consoles (terminals)
  - Sound input and output
  - Special devices (`/dev/null` and `/dev/zero`)
- Block devices
  - Hard and floppy disks, ramdisks
  - CD and DVD
  - Loop devices
- Network interfaces
  - ethernet, atm, wireless, ppp, local loopback

Devices' details

- Block and character devices are reflected in the filesystem via special files
- The devices are identified by
  - Major number (1..255)
  - Minor number (0..255)
- The access to a special file is routed to its driver
- The special files are created with **`mknod`**
- List of used devices in **`/proc/devices`**

Character Devices

- Accessed by means of filesystem nodes
  - Well represented by the stream abstraction
  - Usually accessed sequentially
    - There are cases where it possible to move back and forth
- Also known as raw
- Usually implement **`open`**, **`close`**, **`read`** and **`write`** system calls
- Examples: text console, serial port

Block Devices

- Support random access
- Accessed by means of filesystem nodes
- Data to the hardware can be only transferred in blocks
- The driver hides the block operations
  - The user interface is the same as a character device
- The driver must offer the kernel an additional block-oriented interface, invisible to the user
- Can host a file system: the block-oriented interface enables mounting of file systems
- Often buffered (cached) to improve performance

Network Interfaces

- It can be an hardware device but also a pure software device (e.g.: loopback)
- Communication with the kernel is completely different from that used by char or block drivers
  - It sends and receive data packets driven by the network subsystem of the kernel
  - Not easily mappable to a filesystem node
    - Not a stream-oriented device
      - The device doesn’t see the individual streams but only the data packets
    - A unique name is assigned to them
- Administrative tasks, such as setting addresses, modifying transmission parameters, and maintaining traffic and error statistics

Details

- In other UNIX systems there are raw devices corresponding to each block device
  - Used to control the device or to transfer data
- In Linux this is not required
  - The interface to block and character devices is the same

Major and Minor Numbers

- Major number
  - Identifies the driver associated with the device
  - Used by the kernel at open-time to dispatch execution to the appropriate driver
  - 8-bit number, 0 and 255 are reserved
  - Some are statically assigned to the most common devices; others can be requested to the kernel
- Minor number
  - Allows the driver to identify a single device among the set it controls
  - Other parts of the kernel don’t use it
  - 8-bit number

Major Numbers: Static Allocation

- Manually select an unassigned major number at driver (module) initialization
- The registration function returns 0 if successful, or a negative error code

```
int register_chrdev(unsigned int major, const
char *name, struct file_operations *fops)
```

  - name is the name of the device that will appear in **`/proc/devices`**
  - fops is a pointer to an array of functions pointers
- The list of allocated major numbers can be found in **`Documentation/devices.txt`**
- There is no need to specify the minor number

Example of Static Allocation

- Register the driver
  - # insmod chardev.o
- Create a device
  - # mknod /dev/chardev c 254 0
    - Creates a device named chardev whose major and minor numbers are 254 and 0
- Remove the device
  - # rm /dev/chardev
- Unregister the driver
  - # rmmod chardev

Major Number: Dynamic Allocation

- Choosing a unique number for a new driver can be difficult
- Favourable for private use
  - A fixed major number must be assigned if the driver is meant to be useful to the community at large or to be included in the official kernel tree
- Creating device nodes is more difficult, because the number is not known in advance
  - Loading-on-demand cannot be used since the major number assigned can't be guaranteed to always be the same

How to Dynamically Allocate

- We do not know the device major number
- Have a look at `/proc/devices`
  - It contains the major numbers of all devices actually claimed by some driver
- The registration function is invoked with 0 as the requested major number and returns the allocated number, or a negative error code
- Next we proceed as with static allocation

Example of Dynamic Allocation

- Invoke **`insmod`** with all arguments that were passed and use `./` to specify a pathname, since newer **`modutils`** don't look in `.` by default
- Remove stale nodes
- Give appropriate group/permissions, and change the group
- Not all distributions have staff, some have wheel instead

```
#!/bin/sh
module="chardev"
device="chardev"
mode="664"
/sbin/insmod -f ./$module.o $*||exit 1
rm -f /dev/${device}[0-3 ]
major=`awk "\\$2==\"$module \"{print \\$1}"/proc/devices `
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3
group="staff"
grep '^staff:' //etc/group >/dev/null ||group="wheel"
chgrp $group /dev/${device}[0-3 ]
chmod $mode /dev/${device}[0-3 ]
```

Overview

- Introduction
- Devices
  - Different types of devices
  - Major and Minor numbers
  - Static vs. Dynamic allocation
- Writing drivers
  - Initialization and cleanup
  - FOPS
  - Writing simple character driver

Startup

- Usually a driver searches the system for a device it knows how to drive
  - The search varies from a device to the next one
  - Can be controlled by module parameters
- If it does not find any device it is capable of driving it causes the load to fail
- If it finds such a device, it register itself as the driver of a particular major number
  - It registers as the handler for the interrupt level the device uses (`/proc/devices`, `/proc/net/dev`)
  - Setup commands to the device (`/proc/interrupts`)
  - Kernel messages (`/var/log/messages`)

Initialization

- **`int init_module()`**
  - It is called right after **`insmod`**
  - It performs preliminary initialization
  - It registers the driver and associates it with a Major number
    - The best way to assign a major number is to initialize a variable to DEV_MAJOR (defined in dev.h)
    - The default value is 0: dynamic allocation
    - The user can specify a static major assigning a value for DEV_MAJOR on the **`insmod`** command line

```
int register_chrdev(unsigned int major,const
  char *name,struct file_operations *fops);
```


Allocation in init_module

```
result = register_chrdev(dev_major,"dev",&dev_fops);
if (result < 0){
   printk(KERN_WARNING "dev: can ’t get major
   %d\n",dev_major);
   return result;
}
if (dev_major == 0) dev_major = result; /* dynamic */
```

Cleanup

- `void cleanup_module()`
  - It is called after **`rmmod`**
  - It must release the major number
  - Failure to do so has unpleasant effects
    - Leaves a dangling pointer in the kernel
    - The recovery procedure is difficult
      - A reboot would be easier and recommended

```
int unregister_chrdev(unsigned int major,
    const char *name);
```

  - Remember to remove device files, as they refer to the major number which might be assigned to different drivers

File Operations (1/3)

- The set of operations a driver can perform on the device it manages
- Fixed structure defined in `<linux/fs.h>`
  - An array of pointers to the driver's functions
- We can consider files to be objects and the functions operating on it to be the methods (OO terminology)
- A driver must implement the desired behaviour for the file interface

File Operations (2/3)

- Traditionally called fops, it is an argument to the driver's registration function **`register_chrdev`**
  - Once the driver has been registered in the kernel table, an action on the device file the driver controls causes one of the fops functions to be invoked
    - It must reference a global structure, not to one local to the module's initialization function
- If the driver does not support an operation the pointer must be left NULL
  - The exact behaviour of the kernel when a NULL pointer is specified is different for each function

File Operations (3/3)

- It grows as the functionality of the kernel is expanded
  - New functions are added to the end of the structure
  - Recompiling old drivers places a NULL pointer for the newer operations
  - There is a tagged initialization format that reduces most problems, it is not standard C but an extension specific to the GNU compiler

File Operations: Example

- Tagged initialization syntax

```
struct file_operations my_fops ={
  read:device_read,
  write:device_write,
  open:device_open,
  release:device_release,
  owner:THIS_MODULE,
};
```

  - It makes drivers more portable across changes in the definitions of the structures
  - It makes the code more compact and readable

File Operations

- The following list shows what operations appear in struct **`file_operations`**, in the same order
- The return value of each operation is 0 for success or a negative error code to signal an error, unless otherwise noted

- There are some complex functions that will not be described here

llseek

```
loff_t (*llseek)(struct file *, loff_t,
  int);
```

- It changes the current read/write position in a file
- The new position is returned as a (positive) value
- Errors are signalled by a negative return value
- The `loff_t` is a “long offset” and is at least 64 bits wide even on 32-bit platforms
- If the function is not specified for the driver, a seek relative to end-of-file fails, while other seeks succeed by modifying the position counter in the file structure

read

```
ssize_t (*read)(struct file*, char*,
  size_t, loff_t*);
```

- It retrieves data from the device
- A NULL pointer in this position causes the read system call to fail with –EINVAL (“Invalid argument”)
- A non-negative return value represents the number of bytes successfully read
  - The return value is a “signed size” type, usually the native integer type for the target platform

write and readdir

```
ssize_t (*write)(struct file*, const
  char*, size_t, loff_t *);
```

- It sends data to the device
- If missing, -EINVAL is returned to the program calling the write system call
- The return value, if non-negative, represents the number of bytes successfully written

```
int (*readdir)(struct file *, void *,
  filldir_t);
```

- This field should be NULL for device files
- It is used for reading directories
  - Only useful to filesystems

poll

```
unsigned int (*poll)(struct file *,
  struct    poll_table_struct *);
```

- Is the back-end of two system calls, `poll` and `select`, both used to inquire if a device is readable or writable or in some special state
- Either system call can block until a device becomes readable or writable
- If a driver doesn’t define its poll method, the device is assumed to be both readable and writable, and in no special state
- The return value is a bit mask describing the status of the device

ioctl

```
int (*ioctl)(struct inode *, struct file
  *, unsigned int, unsigned long);
```

- It offers a way to issue device-specific commands
- A few ioctl commands are recognized by the kernel without referring to the fops table
- If the device doesn't offer an ioctl entry point, the system call returns an error for any request that isn't predefined (ENOTTY, "No such ioctl for device")
- If the device method returns a non-negative value, the same value is passed back to the calling program to indicate successful completion

mmap and open

```
int (*mmap)(struct file *, struct
  vm_area_struct *);
```

- It requests a mapping of device memory to a process's address space
- If the device doesn't implement this method, the **mmap** system call returns -ENODEV

```
int (*open)(struct inode *,struct file
  *);
```

- Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method
- If this entry is NULL, opening the device always succeeds, but your driver isn't notified

flush

```
int (*flush)(struct file *);
```

- It is invoked when a process closes its copy of a file descriptor for a device
- It should execute (and wait for) any outstanding operations on the device
- This must not be confused with the `fsync` operation requested by user programs
- Currently flush is used only in the network file system (NFS) code
- If flush is NULL, it is simply not invoked

release and fsync

```
int (*release)(struct inode *, struct
  file *);
```

- This operation is invoked when the file structure is being released
- Like **`open`**, release can be missing

```
int (*fsync)(struct inode *, struct
  dentry *, int);
```

- This method is the back-end of the **`fsync`** system call, which a user calls to flush any pending data
- If not implemented in the driver, the system call returns -EINVAL

fasync and lock

```
int (*fasync)(int, struct file *, int);
```

- It notifies the device of a change in its FASYNC flag
- The field can be NULL if the driver doesn’t support asynchronous notification

```
int (*lock)(struct file *, int,struct
  file_lock *);
```

- It is used to implement file locking
- Locking is a fundamental feature for regular files, but is almost never implemented by device drivers

(*readv) and (*writev)

```
ssize_t (*readv)(struct file*, const
  struct iovec*, unsigned long, loff_t*);
ssize_t (*writev)(struct file *,const
  struct iovec*, unsigned long, loff_t
  *);
```

- Added late in the 2.3 development, they implement scatter/gather read and write operations
- Applications occasionally need to do a single read or write operation involving multiple memory areas; these system calls allow them to do so without forcing extra copy operations on the data

Owner pointer

```
struct module *owner;
```

- This field isn't a method like everything else in the file_operations structure
- It is a pointer to the module that "owns" this structure
- Used by the kernel to maintain the module's usage count

File Operations: Open

- Any initialization in preparation for later operations
  - It increments the usage count
    - The module won’t be unloaded before the file is closed
  - It checks for device-specific errors (such as device-not-ready or similar hardware problems)
  - It initializes the device, if it is being opened for the first time
  - It identifies the minor number and update the f_op pointer, if necessary
  - Allocate and fill any data structure to be put in `filp->private_data`

File Operations: Open

- The module won't be removed unless the usage count is zero
- The kernel maintains the usage count through the owner field of the FOPS
  - Older kernels require modules to do all the work of maintaining their usage count
- The driver never knows the device name, just the major number
- Users can play on this indifference, the following have the same aliasing effect
  - Two special files with same major/minor pair
  - Symbolic or hard links to the special file

File Operations: Release

- The role of the **`release`** method is the reverse of **`open`**
- Sometimes the method implementation is called **`device_close`** instead of **`device_release`**
- Most common tasks
  - It deallocates anything that **`open`** allocated in **`filp->private_data`**
  - It shuts down the device on last close
  - It decrements the usage count
    - The kernel won't be able to unload the module if the usage count doesn't drop to 0

File Operations: Release

- How can the counter remain consistent if a file can be closed without having been opened?
  - The `dup` and `fork` system calls will create copies of open files without calling `open`
    - Each of those copies is then closed at program termination
  - `dup` and `fork` just increment the counter in the existing file structure
  - `release` is called only after the last close
  - `flush` is called at every close

File Operations: Release

- Not every `close` syscall causes the `release` method to be invoked
  - Only the ones that actually release the device data structure
- Neither `fork` nor `dup` creates a new file structure
  - They just increment the usage count in the existing structure
- `close` syscall executes the `release` method only when the counter for the file structure drops to zero
  - The structure is destroyed
- The relationship between the `release` method and the `close` syscall guarantees consistent usage count

FOPS: Read & Write

```
ssize_t read(struct file *filp, char
  *buff, size_t count,loff_t *offp);
ssize_t write(struct file *filp, const
  char *buff, size_t count,loff_t *offp);
```

- `filp` is the file pointer
- `count` is the size of the requested data transfer
- The `buff` argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed
- `offp` is a pointer to a “long offset type” object that indicates the file position the user is accessing
- The return value is a “signed size type”

FOPS: Read & Write

- The main issue associated with the two device methods is the need to transfer data between the kernel address space and the user address space
  - The operation cannot be carried out through pointers in the usual way, or through `memcpy`
- User-space addresses cannot be used directly in kernel space
  - User space memory can be swapped out, and page faults can be generated upon access
  - If the target device is an expansion board instead of RAM, the same problem arises
    - The driver must nonetheless copy data between user buffers and kernel space (and possibly between kernel space and I/O memory)

FOPS: Read & Write

- Cross-space copies are performed by special functions defined in **<asm/uaccess.h>**
  - Such a copy is either performed by a generic function or by functions optimized for a specific data size
- The following kernel functions copy an arbitrary array of bytes and sit at the heart of every **read** and **write** implementation

```
unsigned long copy_to_user(void *to, const
  void from, unsigned long count);
unsigned long copy_from_user(void *to, const
  void *from, unsigned long count);
```


FOPS: Read & Write

- These functions behave like normal **`memcpy`** functions
- The two functions also check whether the user space pointer is valid
  - If the pointer is invalid, no copy is performed
  - If an invalid address is encountered during the copy, only part of the data is copied
  - The return value is the amount of memory still to be copied

FOPS: Read & Write

- If the user pages being addressed are not in memory, the page-fault handler can put the process to sleep while the page is being loaded
- Any function that accesses user space must be reentrant and must be able to execute concurrently with other driver functions
- That’s why semaphores are used to control concurrent access

FOPS: Read & Write

- The file position at `*offp` should be updated to represent the current file position after successful completion of the system call
- Most of the time the `offp` argument is just a pointer to `filp->f_pos`, but a different pointer is used in order to support the `pread` and `pwrite` system calls, which perform the equivalent of `lseek` and `read` or `write` in a single operation

FOPS: Read & Write

- A negative return value indicates the kind of error that occurred
- A return value greater than or equal to 0 tells the calling program how many bytes have been successfully transferred
- If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called

FOPS: Read & Write

- Programs that run in user space always see -1 as the error return value
- The user program needs to access the errno variable to find out what happened, according to **`<linux/errno.h>`**
- The difference is dictated by the POSIX calling standard for system calls and the advantage of not dealing with errno in the kernel

FOPS: Read & Write

- Meaning of the return value:
  - If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred. This is the optimal case
  - If the value is positive, but smaller than count, only part of the data has been transferred. Most often, the application program will retry reading (writing)
    - The library function **`fread`** (**`fwrite`**) reissues the system call until completion of the requested data transfer
  - If the value is 0, end-of-file was reached (nothing was written)
  - A negative value means there was an error
    - Valid errors are those defined in **`<linux/errno.h>`**

FOPS: Readv & Writev

- Vectorized versions of **`read`** and **`write`**
- The input is an array of structures, each of which contains a pointer to a buffer and a length value
- Until version 2.3.44, Linux emulated them with multiple calls to **`read`** and **`write`**
- In many situations, greater efficiency is achieved by implementing **`readv`** and **`writev`** directly in the driver

```
ssize_t (*readv)(struct file *filp, const struct
  iovec *iov, unsigned long count, loff_t *ppos);
ssize_t (*writev)(struct file *filp, const struct
  iovec *iov, unsigned long count, loff_t *ppos);
```


A Simple Character Driver

- The Simple Character Driver (`scad`) counts all data written and read from it and displays the statistics in `/proc/scad`
- Data written to are discarded
- Data read from are a stream of characters

Useful documents

- A. Rubini, Jonathan Corbet – Linux Device Drivers
- Tigran Aivazian – Linux Kernel 2.4 Internals
- Bryan Henderson – Linux Loadable Kernel Modulkes HOWTO
- Ori Pomerantz – Linux Kernel Modules Programming Guide