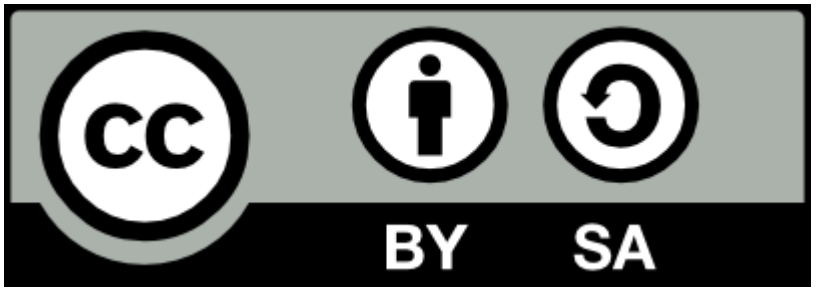
Linux Device Drivers



These slides created by: Karim Yaghmour

Originals at: www.opersys.com/training/linux-device-drivers

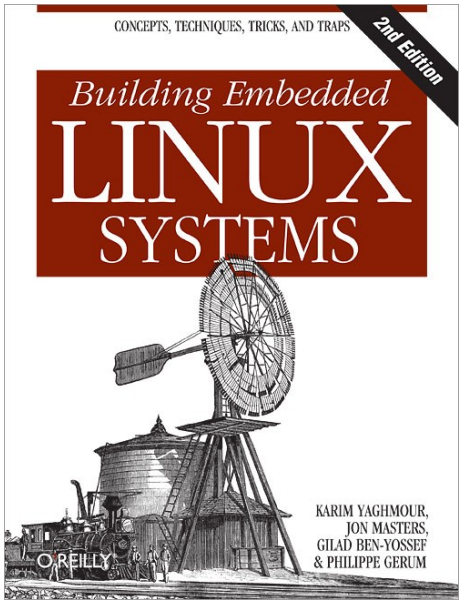
Delivered and/or customized by

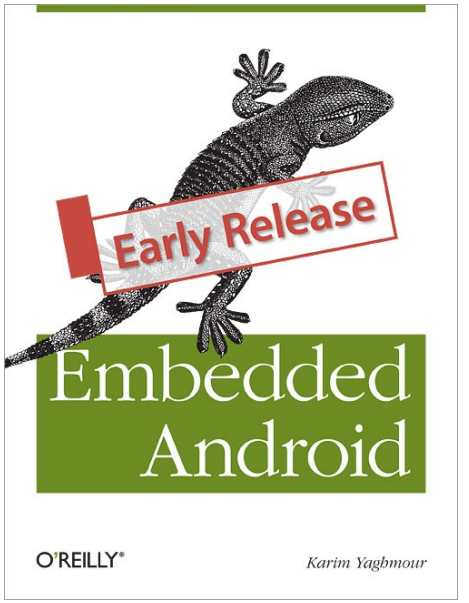
Course structure and presentation

1. About the instructor
2. Goals
3. Presentation format
4. Expected knowledge
5. Day-by-day outline
6. Courseware

1. About the instructor

- Author of:





- Introduced Linux Trace Toolkit in 1999
- Originated Adeos and relayfs (kernel/relay.c)

2. Goals

- To provide an in-depth understanding of how to develop, build and test device drivers for Linux
- To give you a hands-on experience in developing, building and testing device drivers for Linux

3. Presentation format

- Course has two main tracks:
  - Lecture: Instructor presents and discusses material
  - Exercises: Attendees put to practice the material presented with instructor assistance.
- Fast pace. Answers to some questions will be postponed until relevant section is covered.
- Given that there is a lot of material, the instructor will set the speed as required.

4. Expected knowledge

- Basic embedded systems experience
- Basic understanding of operating system concepts.
- C programming experience
- Basic grasp of open source and free software philosophy.
- Good understanding of the debugging process
- Good understanding of complex project architecture.

5. Day-by-day outline

- **Day 1:**
  1. Introduction
  2. Hardware and Linux, a view from user-space
  3. Writing modules
  4. Driver types, subsystem APIs and driver skeletons
  5. Hooking up with and using key kernel resources

- **Day 2:**
  6. Locking mechanisms
  7. Interrupts and interrupt deferal
  8. Timely execution and time measurement
  9. Memory resources
  10. Hardware access

- **Day 3:**
  11. Char drivers
  12. Block drivers
  13. Network drivers
  14. PCI drivers
  15. USB drivers
  16. TTY drivers

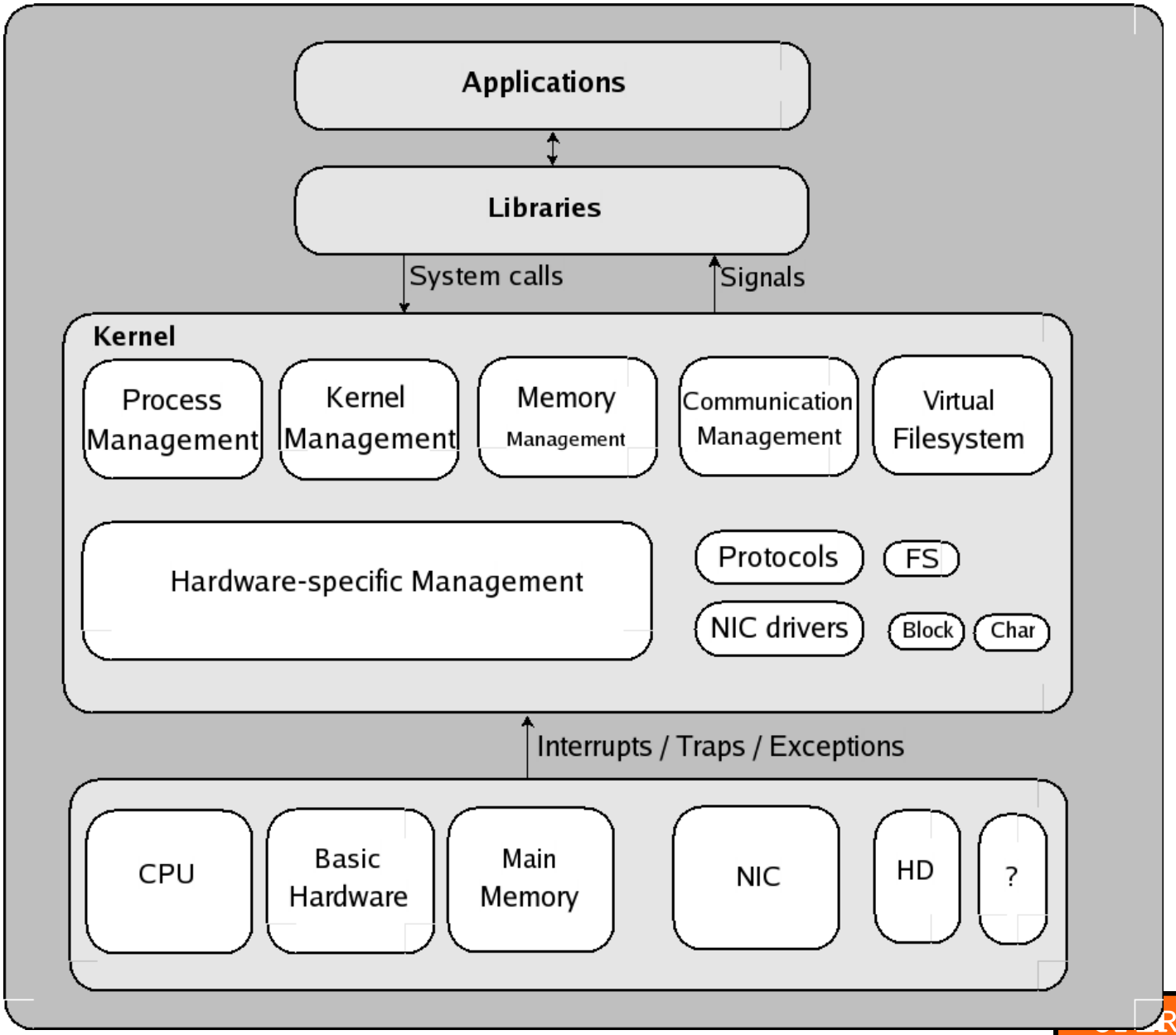
6. Courseware

- “Linux Device Drivers, 3rd ed.” book
- “Device Drivers for Embedded Linux” slides manual.
- Exercise set
- CD-ROM for hands-on session

Introduction

1. System architecture review
2. User space vs. kernel space
3. In the kernel world
4. Drivers
5. Hands-on work environment

1. System architecture review



- Kernel
  - Provide Unix API to applications
  - Manage core I/O
  - Manage memory
  - Control process scheduling
  - etc.
- Drivers
  - Control hardware
  - Interface with kernel
  - Provide expected semantics to kernel

- Libraries
  - Provide “sugar-coated” interfaces to applications
- Applications
  - Talk to kernel through libraries
  - Talk to device through kernel
  - Implement end-user functionality

2. User space vs. kernel space

- Separate address space:
  - No explicit references to objects from other space
- Memory protection amongst processes:
  - No process can directly access or alter other processes' memory areas.
- Memory protection between processes and kernel:
  - No process can access anything inside the kernel
  - Processes that attempt die (segfault)
- Crossing between user space and kernel space is through specific events (will see later)

3. In the kernel world

- Use modules
- Once loaded, core API available:
  - kernel API changes:
    - http://lwn.net/Articles/2.6-kernel-api/
- Have full control ... easy to crash
- Lots of concurrency in the kernel
- Only one “current” process per CPU

4. Drivers

- Built-in vs. modularized
- Userspace drivers?
  - A concept more than a reality
  - X window, libush, gadgetfs, CD writers, ...
  - Hard to map hardware resources (RAM, interrupts, etc.)
  - Slow (swapping, context switching, etc.)
- Security issues
  - Must have parameter checking in drivers
  - Preinitialize buffers prior to passing to uspace

- Licensing reminder
  - Although the use of binary-only modules is widespread, Kernel modules are not immune to kernel GPL. See LDD3, p.11
  - Many kernel developers have come out rather strongly against binary-only modules.
  - Have a look at BELS appendix C for a few copies of notices on binary-only modules.
  - If you are linking a driver as built-in, then you are most certainly forbidden from distributing the resulting kernel under any license other than the GPL.

- Firmware:
  - Often manufacturer provides binary-only firmware
  - Kernel used to contain firmware binaries as static hex strings.
  - Nowadays, firmware loaded at runtime from userspace (like laptop Intel 2200).

5. Hands-on work environment

- Typically would use same cross-development toolchain used for the rest of the system components.
- Qemu
- No actual cross-dev
- Real hardware would require BDM/JTAG
- Build is on regular x86 Linux host

Hardware and Linux, a view from user-space

1. Device files
2. Types of devices
3. Major and minor numbers
4. /proc and procfs
5. /sys and sysfs
6. /dev and udev
7. The tools

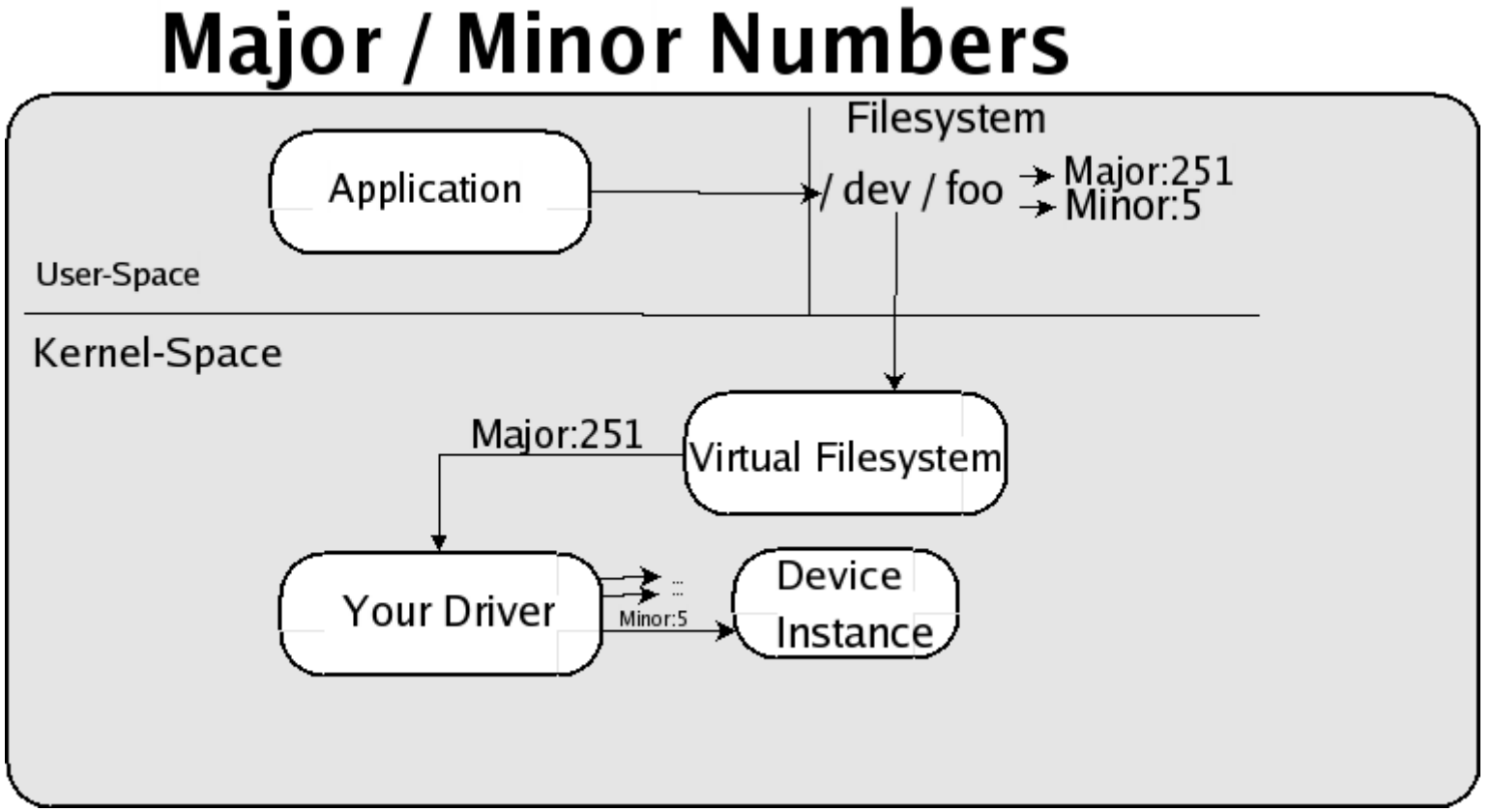
1. Device files

- Everything is a file in Unix, including devices
- All devices are located in the /dev directory
- Only networking devices do not have /dev nodes
- Every device is identified by major / minor number
- Can be allocated statically (devices.txt)
- Can be allocated dynamically
- To see devices present: `$ cat /proc/devices`

2. Types of devices

- What user space sees:
  - Char
  - Block
  - Network
- Abstractions provided by kernel
  - USB
  - SCSI
  - I2C
  - ALSA
  - MTD
  - etc.

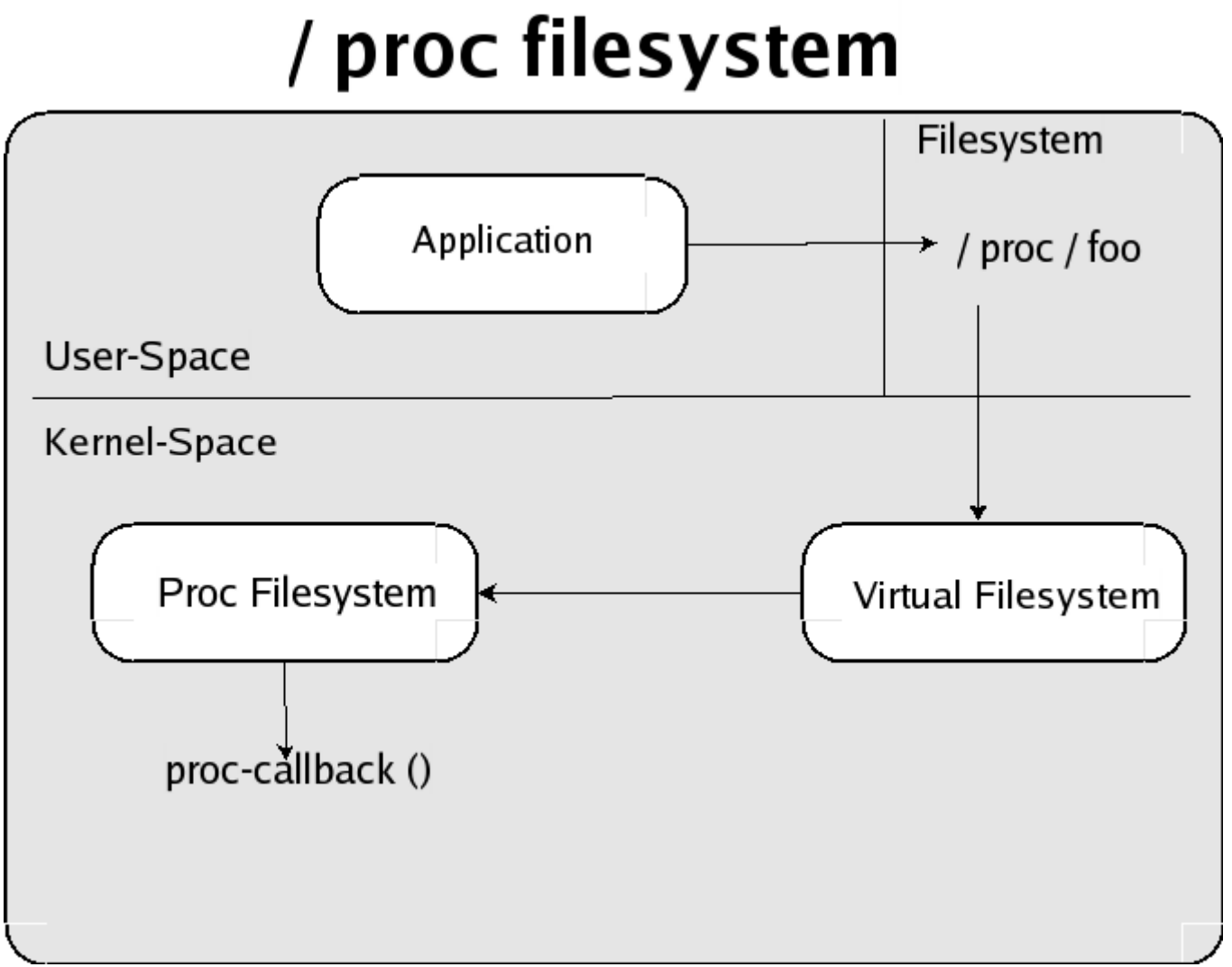
3. Major and minor numbers



- The “glue” between user-space device files and the device drivers in the kernel.
- User-space accesses devices through device “nodes” ... special entries in the filesystem.
- Each device instance has a major number and a minor number.
- Each char and block driver that registers with the kernel has a “major” number.
- When user-space attempts to access a device node with that same number, all accesses result in actions/callbacks to driver.

- Minor number is device “instance”
- Minor number is not recognized or used by the kernel.
- Minor number only used by driver

4. /proc and procfs



- /proc is a virtual filesystem of type “procfs”
- All files and directories in /proc exist only in memory.
- Read / writes result in callback invocation
- /proc is used to export information about a lot of things in the kernel.
- Used by many subsystems, drivers, and core functionality.
- Typically regarded by kernel developers as a mess.

- Example /proc entries:
  - cpuinfo: Info about CPU
  - interrupts: List of interrupts and related drivers
  - iomem: List of I/O memory regions
  - ioports: List of I/O port regions
  - devices: List of active char and block drivers
  - dma: Active DMA channels (ISA)
  - bus/pci: Info tree about PCI
  - bus/usb: Info tree about USB (usbdevfs)

5. /sys and sysfs

- New way of exporting kernel info to user-space
- Meant to eventually replace /proc
- Tightly tied to the kernel's device model
- Provides very good view of hardware and devices tied to it.
- Contains mutiple views of the same hardware:
  - Bus view (/sys/bus)
  - Device view (/sys/devices)
  - Class view (/sys/class)

6. /dev and udev

- /dev is the main repository for device nodes
- Distros used to ship with thousands of entries in /dev; for every possible hardware out there.
- Became increasingly difficult to use as devices were more and more “mobile”.
- With the arrival of sysfs and the related hotplug functionality: udev.
- udev automagically creates the appropriate entries in /dev dynamically.
- Can be configure to provide persistent view

7. The tools

- Many tools to “see” or “control” hardware
- Examples:
  - lspci: list PCI devices
  - lsusb: list USB devices
  - fdisk: partition disk
  - hdparm: set disk parameters
  - ifconfig, iwconfig: configure network interface
  - insmod, modprobe, rmmod, lsmod: manage modules
  - halt, reboot: control system
  - hotplug: manage the adding/removal of hardware

Writing modules

1. Setting up your test system
2. Kernel modules versus applications
3. Compiling and loading
4. The kernel symbol table
5. Preliminaries
6. Initialization and shutdown
7. Modules parameters
8. /sys/modules and /proc/modules

1. Setting up your test system

- Use mainline
- 3rd party kernels can have a different API
- Build system requires targeted kernel

2. Kernel modules versus applications

- init and cleanup
- kspace vs. uspace:
  - Memory protection / isolation
  - Access rights
  - Limited ways for transition:
    - Interrupts
    - Traps
    - Syscalls
- Concurrency
- The "current" process

- Stack limit in kernel: 4K
- Use appropriate allocation primitives to obtain large structures.
- Beware of __fct => internals
- No FP

3. Compiling and loading

- Compiling modules:
  - obj-m := -> module name
  - module-objs:= -> files making up module
  - M= -> module directory (‘pwd’)
  - Example makefile on p.24 of LDD3
  - More complete makefile needs "clean:"
  - Use of kernel build system
- Loading and unloading modules:
  - insmod
  - sys_init_module (kernel/module.c)

- modprobe (resolves unresolved symbols)
- rmmod
- lsmod

- Version dependency:
  - No discussion of modversions
  - Must compile modules against exact kernel used
  - Modules linked to vermagic.o for:
    - Version
    - CPU build flags
  - vermagic tested against targeted kernel at load time

- KERNELDIR specifies version
- linux/modules.h includes linux/version.h which has:
  - UTS_RELEASE => string
  - LINUX_VERSION_CODE => hex
  - KERNEL_VERSION() => macro **for building hex** value for comparison against checkpoint.
- #ifdefs when needed
- Use low-level/high macros to hide details

4. The kernel symbol table

- Basic kernel symbol table exported at build time
- Kernel symbol table kept up-to-date at runtime by kernel.
- Module stacking (ex.: USB, FSes, parport, etc.)
- modprobe resolves dependencies
- EXPORT_SYMBOL():
  - Macro used outside any function scope in module to export symbols for use by other modules.

- EXPORT_SYMBOL_GPL():
  - Same as EXPORT_SYMBOL() but symbols are only available to modules licensed as "GPL".

5. Preliminaries

- Must have:
  - #include <linux/module.h>
  - #include <linux/init.h>
- If need module parameters:
  - #include <linux/moduleparam.h>
- MODULE_LICENSE("...");
  - "GPL v2"
  - "GPL"
  - "GPL and additional rights"
  - "Dual BSD/GPL"

  - "Dual MPL/GPL"
  - "Proprietary"
- MODULE_AUTHOR();
- MODULE_DESCRPITION();
- MODULE_VERSION();
- MODULE_ALIAS();
- MODULE_DEVICE_TABLE();
- Usually, macros put at the end of the file
- To check fields, use modinfo command

6. Initialization and shutdown

- Initialization:
  - Declare init as "static" => file-limited scope
  - __init => drop fct after init
  - module_init => specify module initializtion function
  - Use relevant kernel API to register callbacks or structures with callbacks.
  - Register all sorts of things:
    - devices, filesystems, line disciplines, /proc entries, etc.
- The cleanup function:
  - __exit to specify that cleanup is for unloading only

- void => no return value
- Use module_exit()
- In exit, reverse deallocation order for resources allocated in init.
- No cleanup = no unloading

- Error handling during initialization:
  - Registration/allocation may fail
  - Failure to deallocate on error = unstable
  - Use reverse-order goto instead of per-error rollback
  - Use proper return code in case of error (<linux/errno.h>)

- Use of custom cleanup:
  - To be called from init or as exit
  - Checks for non-null global vars
  - Can't be marked as __exit
- Module-loading races:
  - Callbacks could be invoked as soon as they are registered:
    - Make sure callbacks aren't active prior to load finish
    - Properly code module to complete internal registration prior to callback registration.

- In case of init failure, some kernel parts may already be using fcts registered prior to failure.

7. Modules parameters

- May need to specify some params at load time
- Specified at load time:
  - insmod
  - modprobe (/etc/modprobe.conf)
- module_param(): Used outside of any function scope
  - Name of variable
  - Variable type
  - Permission mask for sysfs

- Var types:
  - bool
  - invbool
  - charp
  - int
  - long
  - short
  - uint
  - ulong
  - ushort

- module_param_array for parameters array:
  - For each entry: name, type, num, and perm
  - Comma separated entries
  - num is number of elements in array
- Use default values for all params
- Defaults change on insmod only if requested by user.
- Permission are as specified in <linux/stat.h>:
  - S_IRUGO => read-only for world
  - S_IRUGO | S_IWUSR => root-only write

- Writable parameters do not generate signal to module, must be detected live.

8. /sys/module and /proc/modules

- /sys/module has one directory per loaded module.
- /proc/modules is older interface providing list of loaded modules.

Types of drivers, subsystem APIs and driver skeletons

1. Char device driver
2. Block device driver
3. Network device driver
4. MTD map file
5. Framebuffer driver

1. Writing a char device driver

- Register char dev during module initialization
- Char dev registration: include/linux/fs.h

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

- First param: Major number
- Second param: Device name (as displayed in /proc/devices)
- Third param: File-ops
  - Defined in include/linux/fs.h
  - Contains callbacks for all possible operations on a char device.

```
struct file_operations {
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned ...
int (*check_flags)(int);
int (*dir_notify)(struct file *filp, unsigned long arg);
int (*flock) (struct file *, int, struct file_lock *);
};
```

- Call *register_chrdev()* and pass it a valid *file_operations* structure.
- Return 0 from initialization function to tell insmod that everything is OK.
- That's it. Every time the device in /dev having the same major number as the one you registered is opened, you driver will be called.
- To remove char dev on rmmod:

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

2. Writing a block device driver

- Register block dev during module initialization
- Block dev registration: include/linux/fs.h

```
int register_blkdev(unsigned int,
                    const char *);
```

- First param: Major number
- Second param: Device name
- Disk allocation: include/linux/genhd.h

```
struct gendisk *alloc_disk(int minors);
```

- Block queue registration: include/linux/blkdev.h

```
extern void blk_init_queue(request_fn_proc *,
                   spinlock_t *);
```

- Queue of pending I/O operations for device

- First param: Queue handler function
- Second param: Lock for accessing queue
- Call *register_blkdev().*
- Call *alloc_disk()* and pass it the number of disks.
- Call *blk_init_queue()* and pass it a valid callback.
- Return 0 from init function to tell insmod status

- Now, all block operations on your device (/dev entry with same major number as driver) will be queued to your driver.
- To remove block dev on rmmod:

```
void blk_cleanup_queue(request_queue_t *);
void put_disk(struct gendisk *disk);
int unregister_blkdev(unsigned int, const char *);
```

3. Writing a network device driver

- Register net dev during module initialization
- Net dev registration: include/linux/netdevice.h

```
int     register_netdevice(struct net_device *dev);
```

- Param: net device ops
  - Defined in include/linux/netdevice.h
  - Contains all callbacks related to network devices
  - This is a huge structure with A LOT of fields
- Call *register_netdevice()* and pass it a valid *net_device* structure.
- Return 0 as status to insmod

- Your device will need to be *opened* by the kernel in response to an *ifconfig* command.
- Your *open()* function must allocate a packet queue to deal with packets sent to your device.
- Calling your device will depend on packet routing at the upper layers of the stack.
- To remove: `unregister_netdev(struct net_device *dev);`

4. Writing an MTD map file

- Must find device in memory and then register it
- Finding a device in memory: include/linux/mtd/map.h

```
struct mtd_info *do_map_probe(char *name,
                              struct map_info *map);
```

- First param: Type of probe (ex: “cfi_probe”)
- Second param: map-info
  - Defined in include/linux/mtd/map.h
  - Information regarding size and buswidth
  - Functions for accessing chip

```
struct map_info {
char *name;
unsigned long size;
unsigned long phys;
#define NO_XIP (-1UL)

void __iomem *virt;
void *cached;

int bankwidth;
#ifdef CONFIG_MTD_COMPLEX_MAPPINGS
map_word (*read)(struct map_info *, unsigned long);
void (*copy_from)(struct map_info *, void *, unsigned long, ssize_t);

void (*write)(struct map_info *, const map_word, unsigned long);
void (*copy_to)(struct map_info *, unsigned long, const void *, ssize_t);
#endif
void (*inval_cache)(struct map_info *, unsigned long, ssize_t);

/* set_vpp() must handle being reentered -- enable, enable, disable
   must leave it enabled. */
void (*set_vpp)(struct map_info *, int);

unsigned long map_priv_1;
unsigned long map_priv_2;
void *fldrv_priv;
struct mtd_chip_driver *fldrv;
};
```

- Once located, use *add_mtd_partitions()* to provide partition information to MTD subsystem.

- *add_mtd_partition()* is in include/linux/mtd/partitions.h

```
int add_mtd_partitions(struct mtd_info *,
                       struct mtd_partition *,
                       int);
```

- First param: pointer returned by *do_map_probe()*
- Second param: partition information as we saw earlier.
- Third param: number of partitions

5. Writing a framebuffer driver

- Register framebuffer during module init
- Framebuffer registration: include/linux/fb.h

```
int register_framebuffer(struct fb_info *fb_info);
```

- Param: fb-info
  - Defined in include/linux/fb.h
  - Contains callbacks for all framebuffer operations

```
struct fb_info {
int node;
int flags;
struct fb_var_screeninfo var;    /* Current var */
struct fb_fix_screeninfo fix;    /* Current fix */
struct fb_monspecs monspecs;     /* Current Monitor specs */
struct work_struct queue;   /* Framebuffer event queue */
struct fb_pixmap pixmap;    /* Image hardware mapper */
struct fb_pixmap sprite;    /* Cursor hardware mapper */
struct fb_cmap cmap;        /* Current cmap */
struct list_head modelist;      /* mode list */
struct fb_ops *fbops;
struct device *device;
#ifdef CONFIG_FB_TILEBLITTING
struct fb_tile_ops *tileops;    /* Tile Blitting */
#endif
char __iomem *screen_base;  /* Virtual address */
unsigned long screen_size;  /* Amount of ioremapped VRAM or 0 */
void *pseudo_palette;       /* Fake palette of 16 colors */
#define FBINFO_STATE_RUNNING    0
#define FBINFO_STATE_SUSPENDED  1
u32 state;              /* Hardware state i.e suspend */
void *fbcon_par;                /* fbcon use-only private area */
/* From here on everything is device dependent */
void *par;
};
```

- Call *register_framebuffer()* and pass it a valid *fb_info* structure.
- Return 0 from init code
- Access the /dev/fbX (where X is your framebuffer's registration order in relationship to other fb drivers) results in the functions provided in *fb_info* to be called.
- To remove fb dev on rmmod:

```
int unregister_framebuffer(struct fb_info *fb_info);
```

Hooking up with and using key kernel resources

1. printk
2. /proc
3. Introduction to sysfs
4. Sysfs entry types
5. Sysfs layout
6. Kobjects, ksets, and subsystems
7. Low-level sysfs operations
8. Hotplug event generation
9. Buses, devices, and drivers

10. Classes
11. Sysfs example
12. Hotplug
13. Copy to/from user
14. Dealing with firmware

1. printk

- Basics
  - Equivalent to libc's printf()
  - Same semantics as printf
  - Recognizes 8 loglevels:
    - KERN_EMERG:
      - Very urgent messages, right before crashing
    - KERN_ALERT:
      - Alert requiring immediate action
    - KERN_CRIT:
      - Critical issue
    - KERN_ERR:
      - Important error

- KERN_WARNING:
  - Non-critical warnings
- KERN_NOTICE:
  - Normal, but worth noting.
- KERN_INFO:
  - Informational
- KERN_DEBUG:
  - Debugging messages

- No loglevel => DEFAULT_MESSAGE_LOGLEVEL => usually KERN_WARNING
- Print to console depends on loglevel (console_loglevel)

- When klogd and syslogd are running => append to /var/log/messages, regarldess of value of console_loglevel.
- klogd does not save consecutive identical lines, just their count.
- If klogd not running, must manually read /proc/kmsg.
- console_loglevel set to DEFAULT_CONSOLE_LOGLEVEL.
- console_loglevel set through sys_syslog().
- Read loglevel config from /proc/sys/kernel/printk LDD3 (p.77).

- Write to /proc/sys/kernel/printk modifies current loglevel.

- Redirecting console messages:
  - May send messages to specific virtual console
  - By default, messages sent to current virtual terminal
  - Can use ioctl(TIOCLINUX) to direct messages to a given console
  - See TIOCLINUX in drivers/char/tty_io.c for more info.
- How messages get logged:
  - Circular buffer

- Size is __LOG_BUF_LEN, configurable at build time from 4 KB to 1MB
- Use of sys_syslog or /proc/kmsg to read content
- printk wakes up whoever is waiting on either
- sys_syslog may be made not to consume
- /proc/kmsg is always consuming (klogd), like a FIFO.
- Wrap-around on overflow
- Data is unprocessed if unread

- klogd dispatches messages to syslogd, which checks /etc/syslog.conf for figuring out how to deal with such messages.
- syslogd logs messages by facility, kernel is LOG_KERN.
- May want to customize /etc/syslog.conf for dispatching kernel messages.
- syslogd messages immediately flushed to disk => performance issue.
- syslogd may be made to avoid this by prefixing log filename with hyphen.

- Turning the messages on and off:
  - Use C macros to disable/enable debug statements at build time.
  - See LDD3 p.80 for example.
- Rate limiting:
  - Too many printks:
    - Printk buffer overflow
    - Too many messages to console = slow
    - Not professional on production systems
    - etc.

- Use:

```
if(printk_ratelimit())
    printk(...);
```

  - If too many messages to console, printk_ratelimit() retval != 0
  - Rate customizable through /proc/sys/kernel/printk_ratelimit
- Printing device numbers:
  - Macros for pretty-formatting device number (<linux/kdev_t.h>):
    - int **print_dev_t**(char *buffer, dev_t dev);
    - char ***format_dev_t**(char *buffer, dev_t dev);

- Difference in return val (quantity vs. buffer)
- buffer of 20+ bytes

2. /proc

- Avoiding syslog overhead
- Using the /proc filesystem:
  - int (*__read_proc__)(char *page, char **start, off_t offset, int count, int *eof, void *data);
  - struct proc_dir_entry *__create_proc_read_entry__(const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void *data);
  - seqfiles:
    - For easy implementation of large entries in /proc
    - <linux/seq_file.h>

- Prototypes:
  - void ***start**(struct seq_file *sfile, loff_t *pos);
  - void ***next**(struct seq_file *sfile, void *v, loff_t *pos);
    - v is iterator as returned by previous start() or next()
    - pos is file position
  - int **show**(struct seq_file *sfile, void *v);
  - void **stop**(struct seq_file *sfile, void *v);
- Sequence of calls from start to stop is atomic.
- Functions to be used by show():
  - int **seq_printf**(struct seq_file *sfile, const char *fmt, ...);
    - Equivalent of printf
    - If retval > 0, buffer has filled and output is discarded
    - Usually retval is ignored by users of seq_printf()
  - int **seq_putc**(struct seq_file *sfile, char c);
    - Equivalent of userspace putc()

- int **seq_puts**(struct seq_file *sfile, const char *s);
  - Equivalent of userpace puts()
- int **seq_escape**(struct seq_file *m, const char *s, const char *esc);
  - Print "esc" characters found in "s" in octal form
  - Typical value for "esc": " \t\n\\"
- int **seq_path**(struct seq_file *sfile, struct vfsmount *m, struct dentry *dentry, char *esc);
  - Print out filename associated with directory entry
  - Not usually used in drivers.

- Declare functions in struct seq_operations:

```
static struct seq_operations my_seq_ops = {
        .start = my_seq_start,
        .next  = my_seq_next,
        .stop  = my_seq_stop,
        .show  = my_seq_show
};
```

- Declare file-ops for connecting seq-ops:

```
static struct file_operations my_proc_ops =  {
        .owner    = THIS_MODULE,
        .open     = my_proc_open,
        .read     = seq_read,
        .llseek   = seq_lseek,
        .release  = seq_release
};
```

  - seq_read, seq_lseek, seq_release provided by kernel
  - my_proc_open:

```
static int my_proc_open(struct inode *inode, struct
  file *file)
{
        return seq_open(file, &my_seq_ops);
}
```

- Register entry with /proc:

```
entry = create_proc_entry("scullseql", 0, NULL);
if (entry)
    entry->proc_fops = &scull_proc_ops;
```

3. Introduction to sysfs

- Need a way to organize the system layout for various kernel parts to use
- Existing device model used for:
  - Power management and system shutdown:
    - Devices shut down from leaves to trunk
  - Communications with user space:
    - Device layout visible through sysfs
    - Devices tunable through sysfs entries
  - Hotpluggable devices:
    - Sending info regarding newly-plugged devices

- Device classes:
  - Tell apps which types of peripherals are present
- Object lifecycles:
  - Objet relationships and ref counting
- Device model tree can be very complex (see /sys)
- Self-coherent model
- Driver writers need not worry too much about bookeeping.
- Typically, subsystems take care of sysfs entries and kobject hierarchies.

4. Sysfs entry types

- Directories: ksets or kobjects
- Files: kobject attributes
  - Attribute types:
    - Default (kset-specific list of attributes)
    - Nondefault (attributes specific to kobject)
    - Binary
- Symlinks: Relationships between kobjects

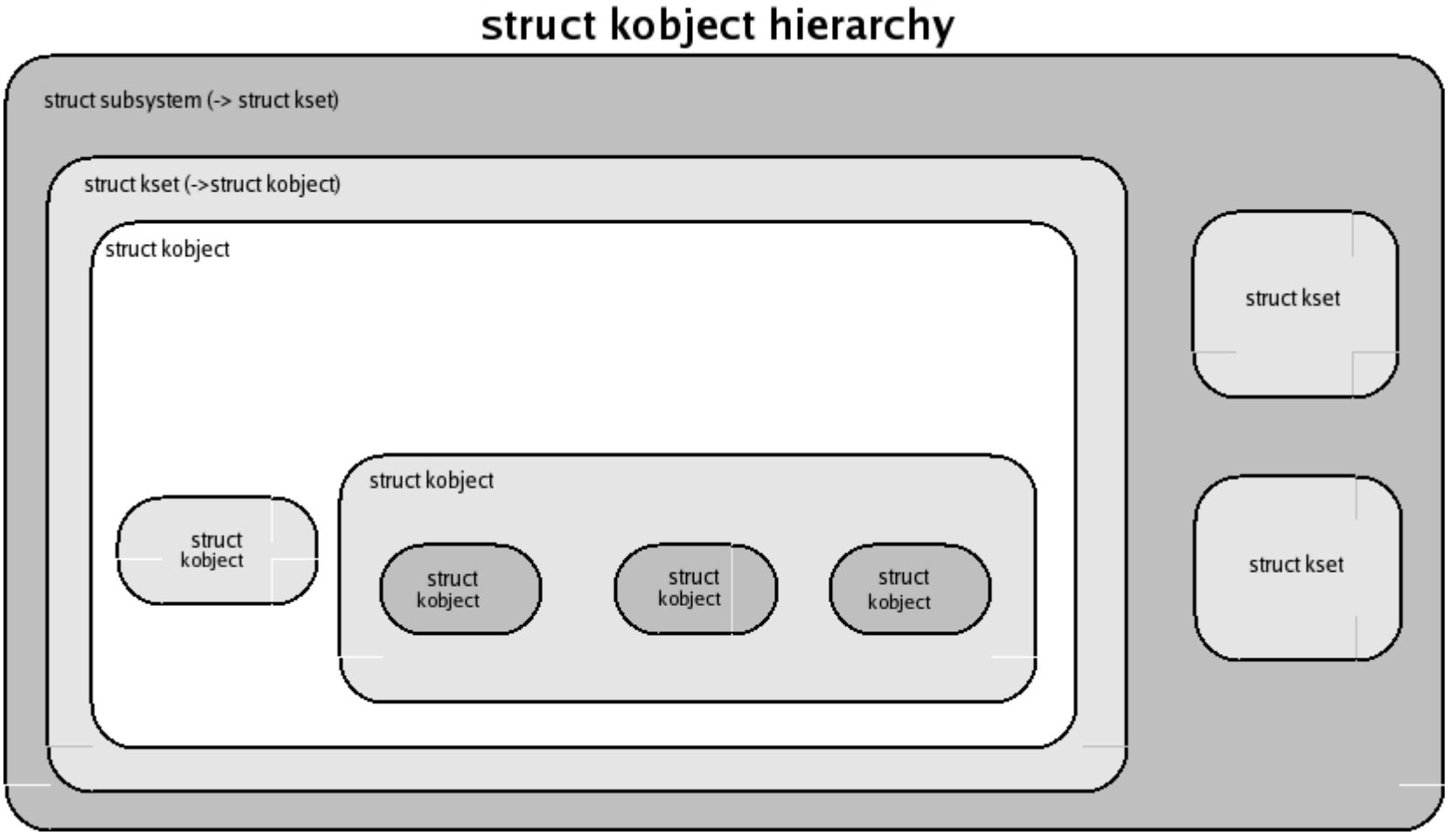
5. Sysfs layout

- /sys/devices:
  - Actual/real device tree starting from CPU
- /sys/bus:
  - Per-bus view of the system
  - Each bus, regardless of how it connects to other buses, has its own separate entry in /sys/bus.
  - Each bus entry has at least two subdirectories:
    - devices: contains one directory for each device connected to the bus.
    - drivers: contains one directory for each driver loaded for the bus.

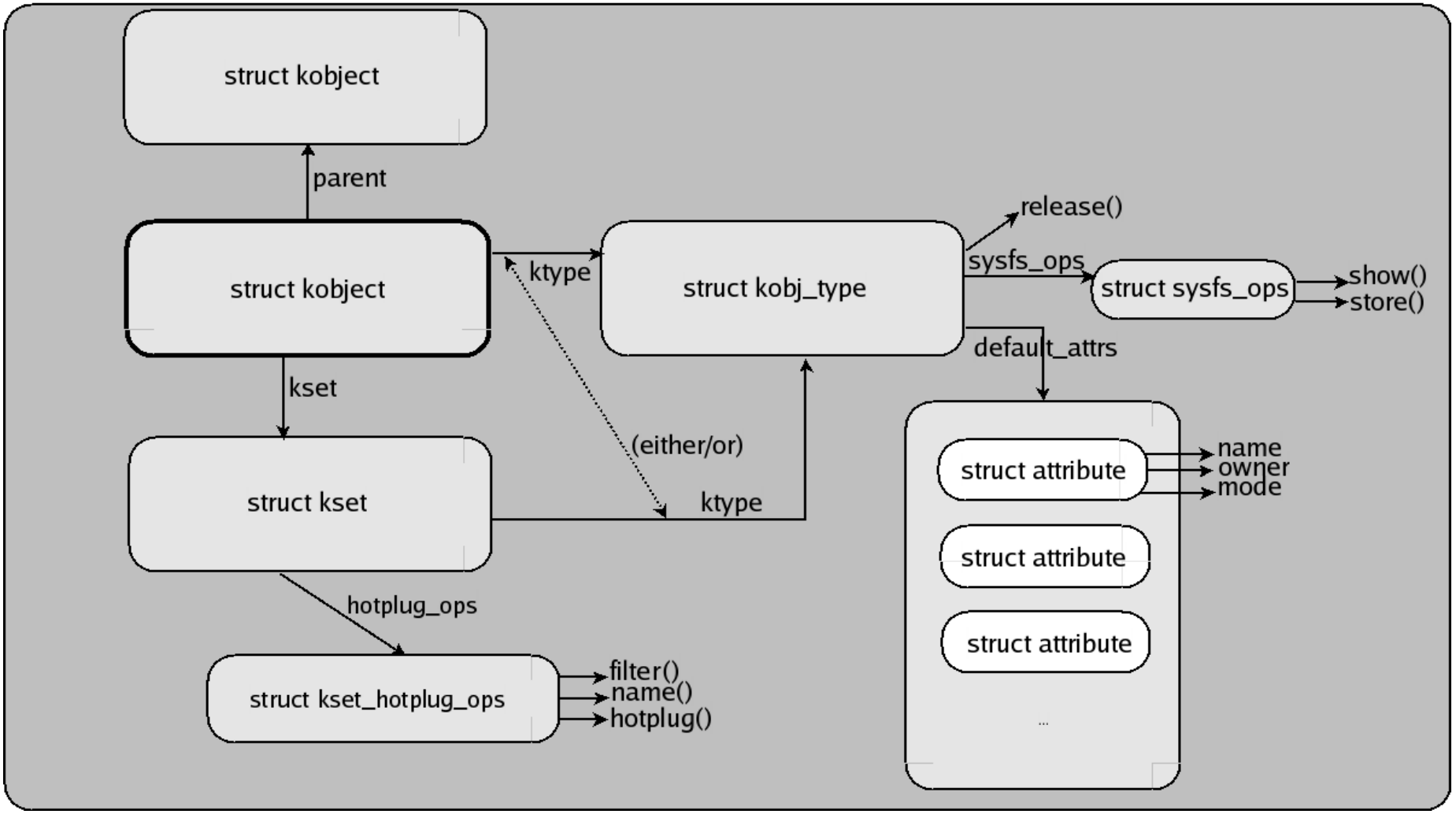
- /sys/class:
  - Class-view of the hardware in the system
  - Each entry is a different type of device
  - Each device-type entry contains list of devices of the given type available in the system.
  - Devices are grouped by type, regardless of the type of bus they are connected to.
- /sys/block:
  - Special directory for the "block" class
- /sys/firmware:
  - Information exported by the system firmware

- /sys/kernel:
  - Information exported by the kernel
- /sys/module:
  - Module view
  - Contains one entry for each module loaded in the system
  - Each entry contains a refcount and a directory containing the sections exported by the module which could be of interest to user-space.
- /sys/power:
  - Information pertaining to power management.

6. Kobjects, ksets, and subsystems



struct kobject



- Introduction:
  - kobject is smallest element of device model
  - struct kobject: <linux/kobject.h>
    - Refcounting
    - Sysfs visibility (though not all kobjects displayed in sysfs)
    - C structure glue
    - Hotplug event handling
- Kobject basics:
  - Embeddeding kobjects:
    - kobjects always tied to something else
    - Almost never exist independently

- Typically part of another struct {}
- Use container_of() to retrieve parent struct
- cdev example:

  struct cdev *device = container_of(kp, struct cdev, kobj);

- kobject

```
struct kobject {
        char *k_name;
        char name[KOBJ_NAME_LEN];
        struct kref kref;
        struct list_head entry;
        struct kobject *parent;
        struct kset *kset;
        struct kobj_type *ktype;
        struct dentry *dentry;
};
```

- kobject initialization:
  - 1- Must memset() kobject instance to 0. Otherwise serious crashes.

- 2- Initialize object and set refcount to 1:
  - void **kobject_init**(struct kobject *kobj);
- 3- Set object's name:
  - int **kobject_set_name**(struct kobject *kboj, const char *format, ...);
  - Like printf
  - Check retval as function may fail

- Reference count manipulation:
  - struct kobject ***kobject_get**(struct kobject *kobj);
    - Increment ref count
    - retval is kobject ptr if success
    - retval is NULL if object is being destroyed
  - void **kobject_put**(struct kobject *kobj);
    - Decrement ref count
    - Call at least once to match kobject_init()

- Release functions and kobject types:
  - What happens when refcount goes to 0
  - Asynchronous event (not necessarily predictable)
  - Must provide "release" method for every kobject
  - Release method associated with struct kobj_type, not kobject:

```
struct kobj_type {
        void (*release)(struct kboject *);
        struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
}
```

  - Every kobject has one kobj_type => ktype
  - If kobject not part of kset, ktype will contain actual release method.
  - If kobject part of kset, kset will provide the struct kobj_type with the release method.

    - Get proper kobj_type:
        - struct kobj_type ***get_ktype**(struct kobject *kobj);
- Kobject hierarchies, ksets, and subsystems:
    - Basics:
        - Need to tie kobjects together according to subsystem structure
        - Possible ties:
            - "parent" ptr in kobject:
                - Dictates sysfs layout
            - ksets
    - Ksets:
        - Object container / aggregation

- struct kset: <linux/kobject.h>

```
struct kset {
        struct subsystem *subsys;
        struct kobj_type *ktype;
        struct list_head list;
        spinlock_t list_lock;
        struct kobject kobj;
        struct kset_hotplug_ops *hotplug_ops;
};
```

- Is a kobject itself (contains kobject in struct)
- Each kset has separate sysfs directory
- Adding kobject to kset:
  1. Set kobject's kset field to appropriate struct kset
  2. int **kobject_add**(struct kobject *kobj);
     - Check retval for error
     - Object ref count incremented
- Helper function for kobject_init() and kobject_add():
  - **int kobject_register**(struct kobject *kobj);

- Removing **kobject** from kset:
  - void **kobject_del**(struct kobject *kobj);
- Helper function for kobject_del() and kobject_put():
  - void **kobject_unregister**(struct kobject *kobj);
- Relationship summary:
  - kset maintained linked list of embedded kobjects
  - "parent" entry in kobject points to kset->kobject

- Operations on ksets:
  - Basic kset manipulation:
    - void **kset_init**(struct kset *kset);
    - int **kset_add**(struct kset *kset);
    - int **kset_register**(struct kset *kset);
    - void **kset_unregister**(struct kset *kset);

- kset refcounting:
  - struct kset ***kset_get**(struct kset *kset);
  - void **kset_put**(struct kset *kset);
- kset's name is in its kobject struct entry
- ksets have pointer to "struct kobj_type"
- kobjects embedded in kset use the set's "struct kobj_type" instead of having their own.
- kset contains pointer to subsystem
- kset must belong to subsystem

- Subsystems:
  - High-level kernel abstractions
  - Typically, each subsystem has toplevel sysfs entry
  - Should almost never have to create own subsystem

- Subsystem can contain multiple ksets

```
struct subsystem {
        struct kset kset;
        struct rw_semaphore rwsem;
}
```

- Subsystem rwsem used to serialize access to kset's kobject list
- Subsystem declaration macro:
  - **decl_subsys**(name, struct kobj_type *type, struct kset_hotplug_ops *hotplug_ops);
  - Creates a struct subsystem
  - Initializes kset with "type"
  - Actual subsystem name is aggregate of name and "_subsys"
- Subsystem helper functions:
  - void **subsystem_init**(struct subsystem *subsys);
  - int **subsystem_register**(struct subsystem *subsys);

- void **subsystem_unregister**(struct subsystem *subsys);
- struct subsystem ***subsys_get**(struct subsystem *subsys);
- void **subsys_put**(struct subsystem *subsys);

7. Low-level sysfs operations

- Basics:
  - Sysfs is virtual filesystem built on top of kobjects
  - "Attributes" exported by kobjects appear in sysfs as files.
  - Kobjects appear in sysfs when kobject_add() is called
  - Sysfs entry creation:
    - kobjects appear in sysfs as directories with attributes
    - kobject directory name is assigned using kobject_set_name()
    - Directory hierarchy matches kobject/kset/subsystem hierarchy.

- Default attributes:
  - Reminder:

```
struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
};
```

  - "default_attrs" is array of sysfs attribute pointers
  - struct attribute

```
struct attribute {
        char *name;
        struct module *owner;
        mode_t mode;
};
```

    - "name": name of attribute as seen in sysfs
    - "owner": owner module

- "mode": file access mode in sysfs

- Attributes implemented by sysfs_ops:
  - struct sysfs_ops

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *kobj,
        struct attribute *attr,
        char *buffer);
        ssize_t (*store)(struct kobject *kobj,
        struct attribute *attr,
        const char *buffer,
        size_t size);
};
```

  - read() on sysfs attribute generates call to show()
    - Put data in buffer (size is PAGE_SIZE)
    - Return amount of data written
    - Convention: each attribute is one humanly-readable attribute
    - Split multiple information pieces accross multiple attributes

- write() on sysfs attribute generates call to store()
  - Read data from buffer (size max is PAGE_SIZE)
  - Return amount of data decoded
  - Negative retaval means error
  - Data is from uspace and must be validated

- Nondefault attributes:
  - Default attributes usually enough
  - Adding attribute to kobject:
    - int **sysfs_create_file**(struct kobject *kobj, struct attribute *attr);
    - retval is 0 on success
    - retval is < 0 on error

    - Must make sure show() and store() functions know how to deal with attribute
  - Removing attribute:
    - int **sysfs_remove_file**(struct kobject *kobj, struct attribute *attr);
    - Uspace app may still have ref on kobj => show/store may yet get called
- Binary attributes:
  - Sometimes need more than just humanly-readable entries.
  - Ex.: loading/unloading firmware

- struct bin_attribute

```
struct bin_attribute {
        struct attribute attr;
        size_t size;
        ssize_t (*read)(struct kobjet *kobj, char *buffer,
        loff_t pos, size_t size);
        ssize_t (*write)(struct kobject *kobj, char *buffer,
        loff_t pos, size_t size);
};
```

  - "attr": attribute as defined earlier
  - "size": maximum size of attribute (0 for no max)
  - "read"/"write": chardev-like callbacks, one page max per call.
  - End-of-file should be determined by read/write callbacks, no way for sysfs to notify.

- Create/destruction:
  - int **sysfs_create_bin_file**(struct kobject *kobj, struct bin_attribute *attr);
  - int **sysfs_remove_bin_file**(struct kobject *kobj, struct bin_attribute *attr);
- Symbolic links:
  - Basic kobject relationships do not, by themselves provide full picture of how things are tied together in the kernel.
  - Sometimes need to create symbolic links betwen kobjects to show relationships.

- Create/destroy symbolic links:
  - int **sysfs_create_link**(struct kobject *kobj, struct kobject *target, char *name);
  - void **sysfs_remove_link**(struct kobject *kobj, char *name);
- Code creating symlink should be able to tie-in to object linked to in order to remove link when remote object ceases to exist.
- Example of object tying needing symbolic links:
  - There is no way for linking a driver to the device it controls.

- /sys/bus contains entries for each bus. Each of these entries contains at least 2 entries: "devices" and "drivers":
  - The entries in the "devices" directories are symlinks to the actual devices, which are themselves stored in /sys/devices.
  - There is one entry in the "drivers" directories for each driver in the system. Those driver entries contain a symlink back to /sys/devices when the driver is active.
- /sys/devices entries themselves contain symbolic links back to the pertinent /sys/bus entries.

8. Hotplug event generation

- Basics:
  - Method for kernel to notify user-space that a new device has appeared in system (i.e. a new kobject has been added).
  - New devices appear when new kobject has been added or removed:
    - kobject_add()
    - kobject_del()
  - Notifications generate invocation of user-space /sbin/hotplug.
  - /sbin/hotplug may do a number of things, including:
    - Loading a driver

    - Creating a device node
    - Mounting partitions
  - To help uspace hotplug do its job, kobjets (or rather ksets) can provide further information using the proper abstractions.
- Hotplug operations:
  - In-kernel hotplug event handling done using struct kset_hotplug_ops.
  - Within struct kset, there is a hotplug_ops of type struct kset_hotplug_ops.

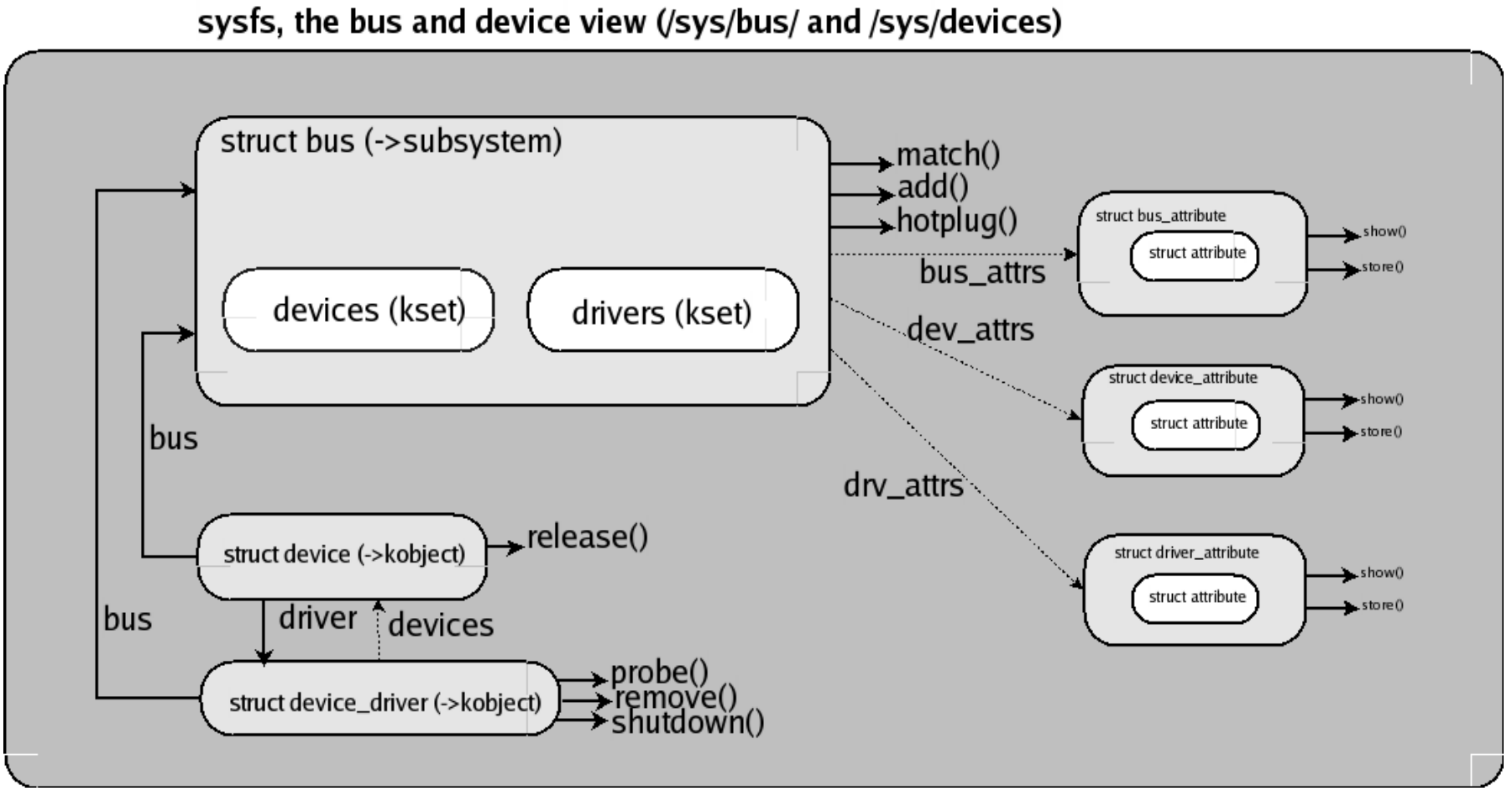
- For a given kobject, kernel traverses kobject parenting until it finds one that has a kset parent with hotplug_ops.
- struct kset_hotplug_ops

```
struct kset_hotplug_ops {
        int (*filter)(struct kset *kset, struct kobject *kobj);
        char *(*name)(struct kset *kset, struct kobject *kobj);
        int (*hotplug)(struct kset *kset, struct kobject *kobj,
        char **envp, int num_envp, char *buffer,
        int buffer_size);
};
```

  - "filter": Called when kernel intends to generate event for "kobj". Allows kset to decide whether an event should indeed be generated. If retval 0, no event generated.

- "name": The subsystem name passed to the user-space hotplug. This is the only parametere passed to /sbin/hotplug.
- "hotplug": This is how /sbin/hotplug knows the rest of the story of what it needs to do. This functions allows kset code to set up environment variables for /sbin/hotplug to use.
  - "envp": Array of environment variables (NULL terminated)
  - "num_envp": Number of environment variables
  - "buffer": Where environment variables are stored
  - retval should be 0
  - Non-zero retval will abort event
  - Hotplug is usually handled by bus driver

9. Buses, devices, and drivers



- Buses:
  - Basics:
    - All devices are plugged into buses
    - "Integrated" devices are part of the "platform" bus
    - Buses can be connected to other buses (USB controller on PCI bus).
    - struct bus_type: <linux/device.h>

```
struct bus_type { /* Most important fields */
        char *name;
        struct subsystem subsys;
        struct kset drivers;
        struct kset devices;
        int (*match)(struct device *dev,
                     struct device_driver *drv);
        struct device *(*add)(struct device *parent,
                              char *bus_id);
        int (*hotplug)(struct device *dev, char **envp,
                       int num_envp, char *buffer,
                       int buffer_size);
};
```

        - "name": Bus name, like "pci" or "usb"
    - Each bus is a subsystem of its own
    - All bus subsystems are part of the "bus" subsystem (/sys/bus)
- Bus registration:
    - Pre-fill struct bus_type
    - Must fill-in "name", "match" and "hotplug"
    - Register:
        - int **bus_register**(struct bus_type *bus);
            - Must check retval
            - On success, can see in /sys/bus
    - Deregister:
        - void **bus_unregister**(struct bus_type *bus);

- Bus methods:
  - match:
    - Called when device or driver added to bus
    - Must determine if given device can be handled by given driver
    - Return non-zero if device can be handled
  - hotplug:
    - Set environment variables for /sbin/hotplug
- Iterating over devices and drivers:
  - In case of having to do an operation for all devices or drivers specified for bus.
  - int **bus_for_each_dev**(struct bus_type *bus, struct device *start, void *data, int (*fn)(struct device *, void *));
    - "start": first device to start from. If NULL, start from first dev on bus.

    - "fn": function to call with dev ptr and "data". If retval is nonzero, iteration stops and bus_for_each_dev() returns retval.
  - int **bus_for_each_drv**(struct bus_type *bus, struct device *start, void *data, int (*fn)(struct device *, void *));
    - Same as buf_for_each_dev()
  - Subsystem rwlock used when these functions are called: careful
- Bus attributes:
  - struct bus_attribute: <linux/device.h>

```
struct bus_attribute {
        struct attribute attr;
        ssize_t (*show)(struct bus_type *bus, char *buf);
        ssize_t (*store)(struct bus_type *bus, const char *buf,
        size_t count);
};
```

  - struct attribute already described

    - show/store already explained as part of struct sysfs_ops
  - Statically creating bus_attribute structures:

    BUS_ATTR(name, mode, show, store);
    - Actual name is "bus_attr_" concatenated with "name"
  - Attribute registration:
    - int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
    - void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
- Devices:
  - Basics:
    - Every device represented using struct device

- struct device: <linux/device.h>

```
struct device { /* Most important fields */
        struct device *parent;
        struct kobject kobj;
        char bus_id[BUS_ID_SIZE];
        struct bus_type *bus;
        struct device_driver *driver;
        void *driver_data;
        void (*release)(struct device *dev);
};
```

  - "parent": Usually the bus controller. Is NULL if topmost device.
  - "kobj": kobject tied to this device
  - "bus_id": Unique device ID on bus
  - "bus": Bus device is attached to
  - "driver": Driver managing device
  - "driver_data": Private driver data
  - "release": Callback for when kobject refcount reaches zero

- Device registration:
  - Minimum struct device fields that need to be set for registration to succeed: parent, bus_id, bus, and release.
  - int **device_register**(struct device *dev);
  - void **device_unregister**(struct device *dev);
  - Buses are devices and, as such, must be registered too
  - If parent is "NULL", successful bus registration results in entry in /sys/devices.

- Device attributes:
  - struct device_attribute: <linux/device.h>

```
struct device_attribute {
        struct attribute attr;
        ssize_t (*show)(struct device *dev, char *buf);
        ssize_t (*store)(struct device *dev, const char *buf,
        size_t count);
};
```

  - Statically creating bus attribute declarations:

    DEVICE_ATTR(name, mode, show, store);

    - Actual name is "dev_attr_" and name
  - Attribute registration:
    - int **device_create_file**(struct device *device, struct device_attribute *entry);
    - void **device_remove_file**(struct device *device, struct device_attribute *attr);

- struct bus_type contains a "dev_attrs" field, which is a list of default attributes for all devices on bus.

- Device structure embeddeding:
  - struct device is often not enough to describe, by itself, the actual device.
  - Most subsystems embed struct device inside their own device structs.
  - Ex: struct pci_dev and struct pci_usb
  - Use container_of() when necessary

- Device drivers:
  - Basics:
    - Object model has device drivers in order to map device drivers to new devices.

- As a side-benefit, some driver config options can be controllered without reference to any specific device.
- struct device_driver: <linux/device.h>

```
struct device_driver { /* Most important fields */
        char *name;
        struct bus_type *bus;
        struct kobject kobj;
        struct list_head devices;
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        void (*shutdown)(struct device *dev);
};
```

  - "name": Driver name as seen in sysfs
  - "bus": Bus type for this driver
  - "kobj": kobject tied to this driver
  - "devices": List of devices currently serviced by driver
  - "probe": Check for the existence of given device
  - "remove": Invoked upon device removal from system
  - "shutdown": Shutdown device

- Registration/deregistation:
  - int **driver_register**(struct device_driver *drv);
  - void **driver_unregister**(struct device_driver *drv);
- Driver attributes:
  - struct driver_attribute: <linux/device.h>

```
struct driver_attribute {
        struct attribute attr;
        ssize_t (*show)(struct device_driver *drv, char *buf);
        ssize_t (*store)(struct device_driver *drv, const char
          *buf,
        size_t count);
};
```

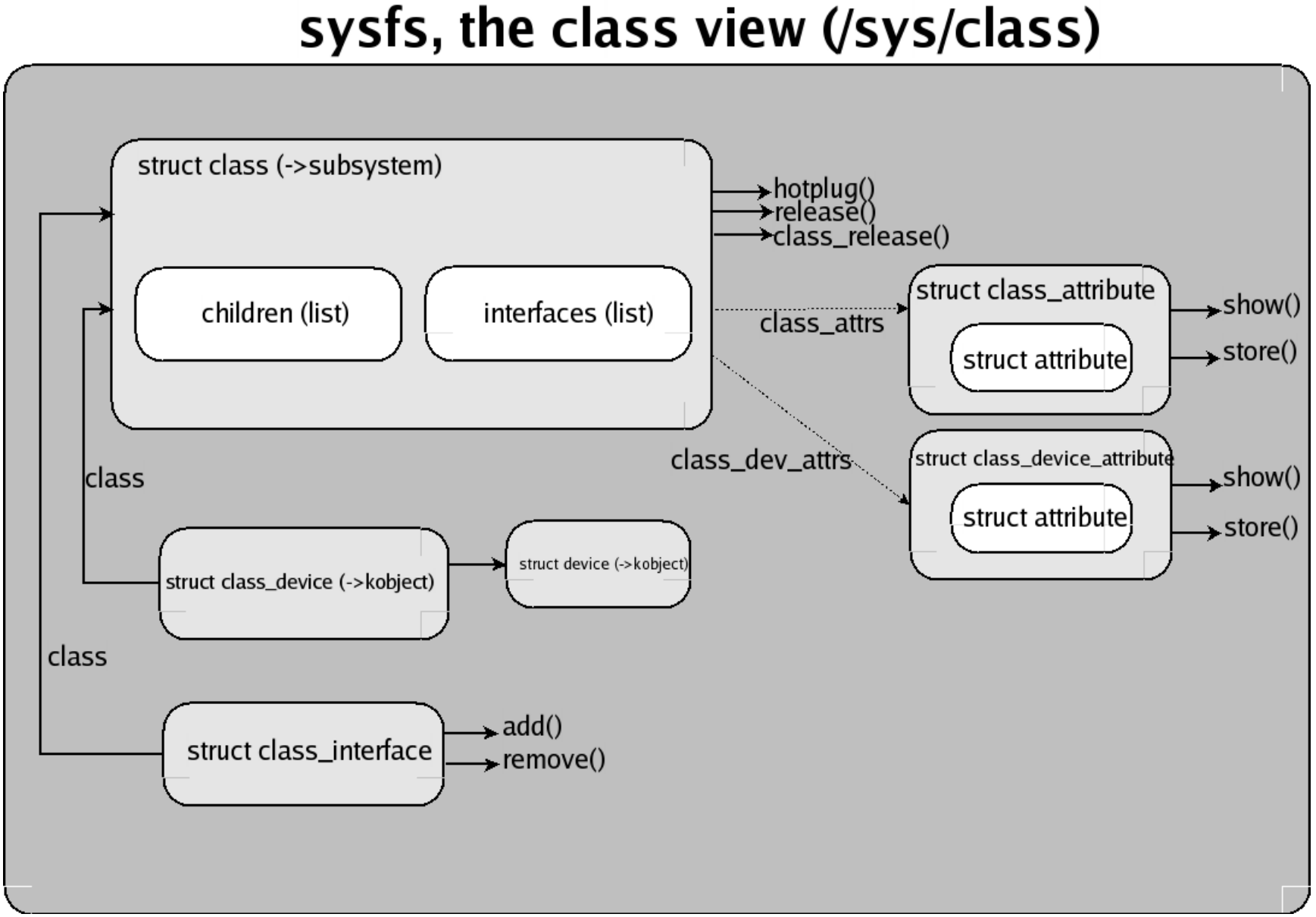
  - Statically creating driver attribute declarations:

DRIVER_ATTR(name, mode, show, store);

- Registering/deregistering attributes:
  - int **driver_create_file**(struct device_driver *drv, struct device_attribute *attr);
  - void **driver_remove_file**(struct device_driver *drv, struct driver_attribute *attr);
- struct bus_type contains a "drv_attrs" field, which is a list of default attributes for all drivers associated with bus.

- Driver structure embedding:
  - Like device struct, driver struct often embedded in other subsystem-specific structs.

10. Classes



- Basics:
  - High-level representation of what is being worked, instead of how it's implemented.
  - Basically a class is an aggregate of all devices of a certain type.
  - See /sys/classes for most classes found on system
  - /sys/block is the only class with its own topmost entry (historical).
  - Class ownership handled by subsystems, no need for driver to care.
  - Drivers should care about classes mainly for exporting data to user- space.

- Interfaces exported by driver core for class manipulation:
  - class_simple
  - Full class interface
- The class_simple interface:
  - Easy-to-use interface for exporting device's assigned ID.
  - struct class_simple ***class_simple_create**(struct module *owner, char *name);
    - Create simple class
    - Must test retval using IS_ERR()

- void **class_simple_destroy**(struct class_simple *cs);
  - Destroy class
- struct class_device***class_simple_device_add**(struct class_simple *cs, dev_t devnum, struct device *device, const char *fmt, ...);
  - Add device to class
  - "devnum": Number assigned to device
  - "device": device's struct device
  - "fmt" and ...: device's name
  - Symlink created to relevant /sys/devices/ entry if device != NULL

- void **class_simple_device_remove**(dev_t dev);
  - Remove "dev" from class
- int **class_simple_set_hotplug**(struct class_simple *cs, int (*hotplug)(struct class_device *dev, char **envp, int num_envp, char *buffer, int buffer_size));
  - Set up a hotplug handler for a class
- The full class interface:
  - Managing classes:

- struct class

```
struct class { /* Most important fields */
        char *name;
        struct subsystem subsys;
        struct list_head children;
        struct list_head interfaces;
        struct class_attribute *class_attrs;
        struct class_device_attribute
          *class_dev_attrs;
        int (*hotplug)(struct class_device *dev, char
          **envp,
        int num_envp, char *buffer, int buffer_size);
        void (*release)(struct class_deivce *dev);
        void (*class_release)(struct class *class);
};
```

  - "release": A device is released from class
  - "class_release": Class is released
- int **class_register**(struct class *cls);
- void **class_unregister**(struct class *cls);

- struct class_attribute

```
struct class_attribute {
        struct attribute attr;
        ssize_t (*show)(struct class *cls, char *buf);
        ssize_t (*store)(struct class *cls, const char *buf,
        size_t count);
};
```

  - CLASS_ATTR(name, mode, show, store);
  - int **class_create_file**(struct class *cls, const struct class_attribute *attr);
  - void **class_remove_file**(struct class *cls, const struct class_attribute *attr);
- Class devices:
  - Classes are device-containers

- struct class_device

```
struct class_device { /* Most important fields */
        struct kobject kobj;
        struct class *class;
        struct device *dev;
        void *class_data;
        char class_id[BUS_ID_SIZE];
};
```

  - "dev": If set, symlink created to corresponding /sys/devices entry.
- int **class_device_register**(struct class_device *cd);
- void **class_device_unregister**(struct class_device *cd);
- int **class_device_renaming**(struct class_device *cd, char *new_name);

- struct class_device_attribute

```
struct class_device_attribute {
        struct attribute attr;
        ssize_t (*show)(struct class_device *cls, char *buf);
        ssize_t (*store)(struct class_device *cls, const char
          *buf,
        size_t count);
};
```

  - CLASS_DEVICE_ATTR(name, mode, show, store);
  - int **class_device_create_file**(struct class_device *cls, const struct class_device_attribute *attr);
  - void **class_device_remove_file**(struct class_device *cls, const struct class_device_attribute *attr);
- Class interfaces:
  - Knowing when devices enter and leave a class

- struct class_interface

```
struct class_interface {
        struct class *class;
        int (*add)(struct class_device *cd);
        void (*remove)(struct class_device *cd);
};
```

- int **class_interface_register**(struct class_interface *intf);
- void **class_interface_unregister**(struct class_interface *intf);

11. Putting it all together

- Looking at how PCI subsystem interacts with object model.
- See figure 14-3, LDD p.392
- Add a device:
  - PCI bus is declared using:

```
struct bus_type pci_bus_type = {
        .name      = "pci",
        .match     = pci_bus_match,
        .hotplug   = pci_hotplug,
        .suspend   = pci_device_suspend,
        .resume    = pci_device_resume,
        .dev_attrs = pci_dev_attrs,
};
```

  - pci_bus_type registered using bus_register() at startup

- Entries created in /sys/pci: devices and drivers
- PCI drivers are struct pci_driver, which contains a struct device_driver.
- When PCI driver registered, struct device_driver is initialized by PCI_code.
- PCI ktype is set to pci_driver_kobj_type
- Driver registered using driver_register()
- When a device is found on the bus, new struct pci_dev created. pci_dev contains a struct device entry.
- After struct pci_dev is initialized, device registered with device_register().

- Device added to list of device in pci_bus_type
- Code then walks list of drivers to find a "match" for device.
- When match found (see earlier explanation), driver's probe() function is invoked to see if driver will accept responsibility for device.
- Once driver acks, driver and device are tied together and the necessary symlinks are created in sysfs.

- Remove a device:
  - Hotplug
  - Removal done through pci_remove_bus_device(), which calls device_unregister().

- device_unregister():
  - Unlinks sysfs entry
  - Removes devices from internal list of devices
  - Calls kobject_del() with device kobject
- Removal of kobject results in hoptlug call
- If last refcount, pci_release_dev() invoked

- Add a driver:
  - pci_register_driver():
    - Initializes struct device_driver in struct pci_driver
    - Calls driver_register()
  - Follow earlier description

    - Match and probe called to match driver with device
- Remove a driver:
    - pci_unregister_driver():
        - Calls driver_unregister()
    - driver_unregister() results in call to release() for each device.
    - Code then waits for all references to driver to be freed before allowing.
    - driver_unregister() to return and, most likely, allow module unloading.

12. Hotplug

- Dynamic devices:
  - Drivers must be able to deal with hardware being plugged and unplugged.
  - Even CPUs and memory hotplug need to be supported.
  - Actual actions to be undertaken depend on bus type
  - See earlier discussion of USB urbs
- The /sbin/hotplug utility:
  - Basics:
    - Called by kernel upon hotplug event
    - Small shell script

- Script invokes scripts from /etc/hotplug.d/
- See "man hotplug" for more details
- Only subsystem name provided to hotplug script
- Actual event details passed using environment variables

- Default environment variables passed to hotplug:
  - ACTION:
    - "add" or "remove" string
  - DEVPATH:
    - Path within sysfs pointing to designated kobject
  - SEQNUM:
    - Each hotplug event has unique sequence number during system lifetime.
  - SUBSYSTEM:
    - Same string passed on hotplug command line

- Subsystem-specific environment variables (see LDD3 for full detail):
  - IEEE1394(FireWire): SUBSYSTEM = "ieee1394"
    - VENDOR_ID
    - MODEL_ID
    - GUID
    - SPECIFIER_ID
    - VERSION
  - Networking: SUBSYSTEM = "net"
    - INTERFACE
  - PCI: SUBSYSTEM = "pci"
    - PCI_CLASS
    - PCI_ID
    - PCI_SUSBSYS_ID
    - PCI_SLOT_NAME

- Input: SUBSYSTEM = "input"
  - PRODUCT
  - NAME
  - PHYS
  - EV, KEY, REL, ABS, MSC, LED, SND, FF
- USB: SUBSYSTEM = "usb"
  - PRODUCT
  - TYPE
  - INTERFACE
  - DEVICE
- SCSI: SUBSYSTEM = "scsi"
  - No specific environment variables
  - There is a SCSI-specific script invoked in user-space
- Laptop docking stations: SUBSYSTEM = "dock"
- S/390 and zSeries: SUBSYSTEM = "dasd"

- Using /sbin/hotplug:
  - Linux hotplug scripts:
    - Try to find driver matching added device
    - Use of maps generated via MODULE_DEVICE_TABLE macros.
    - /lib/module/KERNEL_VERSION/modules.*map
    - Maps for PCI, USB, IEEE1394, INPUT, ISAPNP and CCW.
    - Continue loading all modules relevant found, kernel decides best match.
    - On shut-down, scripts do not remove driver since other devices may have been put under its responpsibility since first load.

- May change in the future as modprobe can read tables generated using MODULE_DEVICE_TABLE without need for modules.*map.

- udev:
  - Allow automatic/easy creation of /dev entries for devices at runtime.
  - Many previous attempts at fixing problem had failed (devfs).
  - Problem with preserving consistent name inside /dev although device might not be connected to system in the same way (USB hd for example.)
  - Notified by /sbin/hotplug of new device addition.

- For proper use by driver, make sure major and minor number attributed to device are exported through sysfs.
- Drivers using existing subsystems need not manually export major and minor numbers through sysfs, as the parent subsystem most likely does that automatically already.
- udev looks for "dev" entry in relevant "/sys/class" path to get major and minor number.
- class_simple is easy way to export "dev"
- "dev" format:
  - <major>:<minor>

13. Copy to/from user

- <asm/uaccess.h>
- functions use architecture-independent "magic" => exception tables.
- unsigned long **copy_to_user**(void __user *to, const void *from, unsigned long count);
- unsigned long **copy_from_user**(void *to, const void __user *from, unsigned long count);
- Similar to memcpy(), but ...

- Can sleep. Code calling should be:
  - Reentrant
  - Capable of executing concurrently with other parts of driver.
  - In a situation where it can sleep
- Retval => amount of memory still to be copied
- If access error, retval != 0

14. Dealing with firmware

- Basics:
  - Some devices need to be loaded with proprietary, closed-source, binary firmware prior to being functional.
  - Hardcoding firmware as hex string in driver is likely GPL violation.
  - Instead, firmware should be loaded from file in userspace.
- The kernel firmware interface:
  - Do not open firmware file and dump to device

- Use appropriate kernel function instead:
  - <linux/firmware.h>
  - int **request_firmware**(const struct firmware **fw, char *name, struct device *device);
    - "name": filename
    - "fw": struct populated by kernel containing pointer to firmware and size of firmware.
    - As everything else from userspace, this firmware must be verified for security reasons.
    - Function *will* sleep
- Having sent firmware to device, it can be released:
  - void **release_firmware**(struct firmware *fw);

- If can't sleep waiting on firmware:
  - int **request_firmware_nowait**(struct module *module, char *name, struct device *device, void *context, void (*cont)(const struct firmware *fw, void *context));
    - "module": THIS_MODULE
    - "context": private date
    - "cont": callback invoked when firmware is available
- How it works:
  - See LDD3, p.407
  - Use of FIRMWARE environment variable to /sbin/hotplug.

Locking mechanisms

1. Concurrency and its management
2. Semaphores and mutexes
3. Completions
4. Spinlocks
5. Locking traps
6. Alternatives to locking
7. Summary

1. Concurrency and its management

- Avoid shared resources when possible (ex. global variables).
- Must "manage" concurrent access whenever resources are shared to guarantee atomicity.
- Use locks or similar mechanisms to implement "critical sections".
- No code instance should use "object" until it is properly initialized for all.
- Must keep track of "object" instances to free when appropriate.

- Linux provides many different mechanisms for implementing critical sections, depending on the situation.

2. Semaphores and mutexes

- P, V, and semaphore int
- Lock with P:
  - if semaphore > 0, decrement and proceed
  - if semaphore <= 0, wait until semaphore is released
- Unlock with V:
  - increment semaphore
  - wake up waiting processes, if any
- When semaphore initially set to "1" => Mutex
- The Linux semaphore implementation:
  - struct semaphore: <asm/semaphore.h>

- Basic initialization:
  - void **sema_init**(struct semaphore *sem, int val);
- Static declare and init mutex:
  - DECLARE_MUTEX(name);
  - DECLARE_MUTEX_LOCKED(name);
- Dynamic init mutex:
  - void **init_MUTEX**(struct semaphore *sem);
  - void **init_MUTEX_LOCKED**(struct semaphore *sem);
  - Typically, use init_MUTEX* prior to doing device registration.

- In Linux:
  - P is "down"
  - V is "up"
- Versions of "down":
  - void **down**(struct semaphore *sem);
    - Decrement and wait as long as necessary
  - int **down_interruptible**(struct semaphore *sem);
    - Decrement and wait, but allow user-space process to continue receiving signals. Must test return value for interruption.
    - If interrupted:
      - Return -ERESTARTSYS if no effect, kernel will restart call
      - Return -EINTR if failure to complete operation
  - int **down_trylock**(struct semaphore *sem);
    - Never wait, try and fail if unable to lock. Must test retval.

- Only one "up":
  - void **up**(struct semaphore *sem);
- Reader/writer semaphores:
  - Allow multiple readers, but only one writer
  - Writer has priority, once one writer asks for semaphore, all readers will wait.
  - Not typical of drivers
  - struct rw_semaphore: <linux/rwsem.h>
  - Initialization:
    - void **init_rwsem**(struct rw_semaphore *sem);

- Read-only access:
  - void **down_read**(struct rw_semaphore *sem);
    - May put task in state TASK_UNINTERRUPTIBLE
  - int **down_read_trylock**(struct rw_sempaphore *sem);
    - Non-zero if access granted.
    - Zero otherwise
  - void **up_read**(struct rw_semaphore *sem);
- Write access:
  - void **down_write**(struct rw_semaphore *sem);
  - int **down_write_trylock**(struct rw_semaphore *sem);
  - void **up_write**(struct rw_semaphore *sem);
  - void **downgrade_write**(struct rw_semaphore *sem);
    - Change a write lock to a read lock.

3. Completions

- "fork"-off task and wait for it to complete
- struct completion: <linux/completion.h>
- Static declaration:
  - DECLARE_COMPLETION(my_completion);
- Dynamic initialization:
  - void **init_completion**(&my_completion);
- Waiting on completion (uninterruptable wait):
  - void **wait_for_completion**(struct completion *c);
- Completion:

- void **complete**(struct completion *c);
  - Wake up just one waiting thread
- void **complete_all**(struct completion *c);
  - Wake up all waiting threads

- Re-initializing for reuse after complete_all():
  - INITIALIZE_COMPLETION(struct completion c);
- Completion in kernel thread:
  - void **complete_and_exit**(struct completion *c, long retval);
  - In the case, for example, where a thread is started in a module and must be killed when module is unloaded.

4. Spinlocks

- Introduction to spinlocks:
  - Most often used mechanism in the kernel
  - Unlike semaphores, can be used in code that cannot sleep.
  - Usually better performance than semaphores.
  - Typically meant for protecting concurent access on SMP systems.
  - Typically defaults to nothing on UP (except for IRQ spinlocks).
  - Either locked or unlocked
  - If lock already taken, spin in tight loop waiting for resource.

- Introduction to the spinlock API:
  - spinlock_t <linux/spinlock.h>
  - Static initialization:
    - spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
  - Dynamic initialization:
    - void **spin_lock_init**(spinlock_t *lock);
  - Enter critical section:
    - void **spin_lock**(spinlock_t *lock);
  - Leave critical section:
    - void **spin_unlock**(spinlock_t *lock);
  - Many more functions

- Spinlocks and atomic context:
  - Never create situation where control may be lost while holding a spinlock, otherwise => deadlock.
  - Code using spinlocks should be atomic
  - Carefully examine which kernel services you call while holding a spinlock (copy_to/from_user, kmalloc, etc. will sleep.)
  - Use special locks if atomic section could be interrupted by an interrupt serviced by a routine requiring that same lock.
  - Hold for as short a time as possible

- The spinlock functions:
  - Locking:
    - void **spin_lock**(spinlock_t *lock);
    - void **spin_lock_irqsave**(spinlock_t *lock, unsigned long flags);
      - Disables interrupts on local CPU and stores previous interrupt state in flags.
      - Takes spinlock (may spin in wait for lock.)
    - void **spin_lock_irq**(spinlock_t *lock);
      - Disables interrupts on local CPU without recording flags
      - Useful if sure no other code has already modified int flag
      - Takes spinlock
    - void **spin_lock_bh**(spinlock_t *lock);
      - Leaves hardware interrupts enabled
      - Disables software interrupts

- 3 levels of "priorities" with spinlocks:
  3- User => spin_lock()
  2- Software interrupt => spin_lock_bh()
  1- Interrupt => spin_lock_irq*()
- Type of lock to use depends on what is the highest priority in which critical section is accessed.
- Unlocking:
  - void **spin_unlock**(spinlock_t *lock);
  - void **spin_unlock_irqrestore**(spinlock_t *lock, unsigned long flags);
    - flags are the same as those passed to spin_lock_irqsave
    - Must be called within same function as spin_lock_irqsave, or will brake on some architectures.

    - void **spin_unlock_irq**(spinlock_t *lock);
    - void **spin_unlock_bh**(spinlock_t *lock);
  - Try locking:
    - int **spin_trylock**(spinlock_t *lock);
    - int **spin_trylock_bh**(spinlock_t *lock);
    - None for interrupt locking
    - Return non-zero on success, and zero otherwise.
- Reader/writer spinlocks:
  - Multiple readers in critical section, only one writer
  - rwlock_t: <linux/spinlock.h>
  - Static initialization:
    - rwlock_t my_rwlock = RW_LOCK_UNLOCKED

- Dynamic initialization:
  - void **rwlock_init**(rwlock_t *lock);
- For readers:
  - void **read_lock**(rwlock_t *lock);
  - void **read_lock_irqsave**(rwlock_t *lock, unsigned long flags);
  - void **read_lock_irq**(rwlock_t *lock);
  - void **read_lock_bh**(rwlock_t *lock);
  - void **read_unlock**(rwlock_t *lock);
  - void **read_unlock_irqrestore**(rwlock_t *lock, unsigned long flags);
  - void **read_unlock_irq**(rwlock_t *lock);
  - void **read_unlock_bh**(rwlock_t *lock);

- For writers:
  - void **write_lock**(rwlock_t *lock);
  - void **write_lock_irqsave**(rwlock_t *lock, unsigned long flags);
  - void **write_lock_irq**(rwlock_t *lock);
  - void **write_lock_bh**(rwlock_t *lock);
  - int **write_trylock**(rwlock_t *lock);
  - void **write_unlock**(rwlock_t *lock);
  - void **write_unlock_irqrestore**(rwlock_t *lock, unsigned long flags);
  - void **write_unlock_irq**(rwlock_t *lock);
  - void **write_unlock_bh**(rwlock_t *lock);

5. Locking traps

- Ambiguous rules:
  - Design for locks from the start
  - Do not call on lock-grabbing functions if already holding a lock that would be acquired by lockee.
  - Properly document functions that expect to get called with locks held.
- Lock ordering rules:
  - If multiple locks are required for implementing some critical sections, always acquire the locks in the same order.
  - Obtain local locks prior to obtaining global ones

- Obtain semaphores prior to obtaining spinlocks
- Avoid multiple-locks whenever possible

- Fine- versus coarse-grained locking:
  - Kernel used to have one big kernel lock (still persists for a very small number of operations.)
  - Most of the kernel's resources now have independent locks.
  - Usually, drivers should start with coarse-locking
  - Avoid overdesigning
  - Use lockmeter to detect contention:

    http://oss.sgi.com/projects/lockmeter

6. Alternatives to locking

- Lock-free algorigthms:
  - Sometimes possible to modify algorithm to obtain lock-free conditions.
  - Ex.: circular buffer with one reader / one writer.
  - Generic circular buffer implementation starting with 2.6.10:
    - <linux/kfifo.h>
- Atomic variables:
  - Usually one integer used for counting
  - atomic_t: <asm/atomic.h>
  - atomic_t holds int on all archs

- Interrupt safe
- SMP-safe
- Cannot count on atomic_t to hold more than 24 bits because of arch details.
- Cannot use atomic_t if multiple arithmetic operations required; would need additional locking.
- Static initialization:
  - atomic_t v = ATOMIC_INIT(0);
- Dynamic initialization:
  - void **atomic_set**(atomic_t *v, int i);
- Reading:
  - int **atomic_read**(atomic_t *v);

- Arithmetic ops without retval:
  - void **atomic_add**(int i, atomic_t *v);
  - void **atomic_sub**(int i, atomic_t *v);
  - void **atomic_inc**(atomic_t *v);
  - void **atomic_dec**(atomic_t *v);
- Arithmetic ops with retval:
  - int **atomic_add_return**(int i, atomic_t *v);
  - int **atomic_sub_return**(int i, atomic_t *v);
  - int **atomic_inc_return**(atomic_t *v);
  - int **atomic_dec_return**(atomic_t *v);
- Arithmetic ops with test:

- int **atomic_sub_and_test**(int i, atomic_t *v);
  - Retval is true if final val is zero, false otherwise
- int **atomic_inc_and_test**(atomic_t *v);
  - Same
- int **atomic_dec_and_test**(atomic_t *v);
  - Same
- int **atomic_add_negative**(int i, atomic_t *v);
  - Retval is true if final val is < 0, false otherwise.

- Bit operations:
  - Bitwise-equivalent of atomic_t
  - Functions declared in <asm/bitops.h>
  - Interrupt-safe
  - SMP-safe

- Fine for setting shared flags
- Tricky to use on critical sections
- Architecture-specific data-typing:
  - Bit to manipulate usually int, but can be unsigned long on some architectures.
  - Address to be modified usually pointer to unsigned long, but can be *void on some architectures.
- Basic ops:
  - void **set_bit**(nr, void *addr);
  - void **clear_bit**(nr, void *addr);
  - void **change_bit**(nr, void *addr);
    - Toggle

- Non-atomic bitval retrieval:
  - int **test_bit**(nr, void *addr);
- Atomic test then modify:
  - int **test_and_set_bit**(nr, void *addr);
  - int **test_and_clear_bit**(nr, void *addr);
  - int **test_and_change_bit**(nr, void *addr);
- Entering critical section:
  - while(test_and_set_bit(nr, addr) != 0) wait_a_little();
  - If already set, this loop will wait, until the other bit of code already holding the lock does a test_and_clear_bit().
  - If multiple threads competing, one of them will have its test_and_set_bit() succeed, and the others will continue looping.

- Leaving critical section:
  - if(test_and_clear_bit(nr, addr) == 0) *ERROR*;
  - If this statement is positive (i.e. we go in the "if"), then something had already released the lock, which means there's a synchronization error somewhere in our code.
- seqlocks:
  - Appropriate for situations where:
    - Resource protected doesn't need to be held for long
    - Resource protected is frequently accessed
    - Write access is rare and fast
  - Readers get "free" access, but must test for collision with writers, and retry in those cases.

- Cannot be used on anything involving pointers.
- seqlock_t: <linux/seqlock.h>
- Static initialization:
  - seqlock_t my_lock = SEQLOCK_UNLOCKED;
- Dynamic initialization:
  - void **seqlock_init**(seqlock_t *lock);
- For readers:
  - Obtain "sequence number":
    - unsigned int **read_seqbegin**(seqlock_t *lock);
  - Conduct simple computation
  - Test if concurrent write occured
    - int **read_seqretry**(seqlock_t *lock, unsigned int seq);

    - If so, discard result and repeat
- Interrupt protected read:
    - unsigned int **read_seqbegin_irqsave**(seqlock_t *lock, unsigned long flags);
    - int **read_seqretry_irqrestore**(seqlock_t *lock, unsigned int seq, unsigned long flags);
- For writers:
    - void **write_seqlock**(seqlock_t *lock);
    - void **write_sequnlock**(seqlock_t *lock);
- Variants for writers:
    - void **write_seqlock_irqsave**(seqlock_t *lock, unsigned long flags);
    - void **write_seqlock_irq**(seqlock_t *lock);

- void **write_seqlock_bh**(seqlock_t *lock);
- void **write_sequnlock_irqrestore**(seqlock_t *lock, unsigned long flags);
- void **write_sequnlock_irq**(seqlock_t *lock);
- void **write_sequnlock_bh**(seqlock_t *lock);

- read-copy-update:
  - Powerful, but complex mechanism
  - Seldom used in drivers
  - Optimized for frequent reads and rare writes
  - Resources protected are accessed via pointers
  - References to resources are atomically protected

- To change data:
  - Make copy of data
  - Change copy
  - Aim relevant pointer to new version
  - When no more refs to old copy exist, kernel frees resource.
- Functions found in <linux/rcupdate.h>
- Reader macros:
  - rcu_read_lock()
    - Disable preemption
  - rcu_read_unlock()
    - Enable preemption

- There is no rcu_write_lock() because of how the algorithm operates, IOW writers are never locking, and that's what makes RCU fast.
- Multiple writers are assumed to synchronize using some other mechanism, such as spinlocks.
- Writer:
  - Allocate new resource
  - Copy data to new resource
  - Change copy
  - Replace pointer seen by read() code
- Complicated part is to know when to free "old copy":

- Other CPUs may still have references to old copy
- Writer must wait until it knows no other instance has pointer to old copy.
- Since all code-paths referencing resource are atomically protected, it is assumed that once every processor on the system has been scheduled at least once, then no other processor still holds a copy of the old data pointer.
- Hence, we can free the old copy.
- The kernel RCU mechanism provides a way for registering a callback to be issued once all processors have been scheduled to clean up the old copy.

- Function to register RCU callback:
  - void **call_rcu**(struct rcu_head *head, void (*func)(void *arg), void *arg);

    - Callback obtains same "arg" as passed to call_rcu().
    - Typically callback issues a kfree().
- Full detail of API and algorithm in <linux/rcupdate.h>

7. Summary

- Semaphores /-> Servicing user-space calls Mutexes
- Completions -> “End” signal shared by
  routines servicing user-space.
- Spinlocks -> SMP systems / disabling interrupts.
- Atomic ops -> Single arithmetic op
- Bit op -> Single bit op
- Seqlocks -> Few writers / lots of readers.
- RCU -> Pointer struct modifications.

Interrupts and interrupt deferal

1. Installing an interrupt handler
2. Implementing a handler
3. Top and bottom halves
4. Interrupt sharing
5. Interrupt-driven I/O

1. Installing an interrupt handler

- The basics:
  - If there's no handler registered for an interrupt, the kernel will ack it but do nothing with it.
  - Kernel keeps track of which driver/handler is associated with an interrupt line.
  - Register interrupt handler:
    - int **request_irq**(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);
    - retval is zero on sucess
    - retval is negative on error (-EBUSY if already allocated).
    - "irq" the IRQ number being requested

- "handler" the IRQ callback
- "flags" bitmask on how to manage interrupt
- "dev_name" name printed out in /proc/interrupts
- "dev_id" pointer needed for handler freeing when interrupt shared. if the interrupt isn't shared, can be set to NULL. Otherwise, set to some unique pointer within driver.

- Interrupt bitmask:
  - SA_INTERRUPT:
    - "Fast interrupt". Execute handler with interrupts disabled on local CPU.
  - SA_SHIRQ:
    - Interrupt can be "shared".

- SA_SAMPLE_RANDOM:
  - Interrupts generated by device can contribute to entropy pool for random number generation (/dev/random and /dev/urandom).
- Unregister handler:
  - void **free_irq**(unsigned int irq, void *dev_id);
- Handler registration can be done on device instantiation or on open().
- Best to register handler on first open() / free on last close().

- The /proc interface:
  - Kernel keeps track of how times each type of interrupt occurs (internal counter).

- Do "cat /proc/interrupts" to see:
  - Interrupt lines that currently have registered handlers
  - The number of times each time of interrupt occured for each CPU.
  - The PIC (Programmable Interrupt Controller) configuration for the interrupt.
  - The driver(s) that have registered handlers for the given interrupt, as provided by the "dev_name" parameter of request_irq().
- Do "cat /proc/stat" and look for the "intr" line to see:
  - The total number of all interrupts that occured since boot
  - The total number of interrupts of a given type that occured since boot, each entry being separated by a space.

- Number of entries in both files will vary greatly between archs.

- Autodetecting the IRQ number:
  - Sometimes interrupt numbers are known in advance
  - Ask PCI config for interrupt number
  - Can ask device to generate interrupt and monitor result.
  - Can't probe shared interrupts
  - Kernel-assisted probing:
    - Only for non-shared interrupts
    - Usually for ISA only

- <linux/interrupt.h>
- unsigned long **probe_irq_on**(void);
  - retval is bitmask of unasigned interrupts
  - record retval for passing to probe_irq_off()
  - Enable interrupts after this call
  - Configure device to emit interrupt
- int **probe_irq_off**(unsigned long);
  - Call to ask kernel which interrupt occured
  - Disable interrupts before this call
  - May need to insert delay prior to calling this function to give time for the interrupt to occur.
  - retval is > 0 if only one interrupt occured
  - retval is 0 if no interrupt occured
  - retval is < 0 if more than one interrupt occured
- Probing can take a lot of time (20 ms for frame-grabber)

- Best to do probing only once at module load time
- Most non-PC platforms don't need probing and above functions are placeholders (including most PPC, and MIPS implementations).

- Do-it-yourself probing:
  - Loop on all possible interrupts
  - Record interrupt handler for a given interrupt
  - Configure device generating an interrupt
  - Wait for interrupt to occur
  - Check to see if handler was called
  - Free interrupt handler

- Fast and slow interrupts:
  - Old kernel abstraction

- Currently, the only difference between handlers is those that use the SA_INTERRUPT flag and those that don't.
- If SA_INTERRUPT is used, handler is called with all local interrupts disabled.
- Most drivers shouldn't use SA_INTERRUPT unless absolutely required.
- The internals of interrupt handling on the x86:
  - Already covered in embedded class
  - Assembly-generating macros in arch/i386/kernel/entry.S push int number on stack and call do_IRQ from arch/i386/kernel/irq.c.

- do_IRQ does:
  - Mask and ack interrupt
  - Requests spinlocks for given IRQ number (to avoid other processors from trying to handle it.)
  - If handler register, call handle_IRQ_event to invoke it
  - Otherwise unlock and return
- handle_IRQ_event:
  - If SA_INTERRUPT not set, reenable interrupts
  - Invoke handler(s)
- Check for scheduling (processes may have been woken up as a result of interrupt).

2. Implementing a handler

- The basics
  - Role:
    - Interact with device regarding interrupt, usually ACK device.
    - Transfer data to/from device as required
    - Wake up processes waiting for device
    - Defer as much work as possible to tasklets or work queues.
  - Restrictions as to what handler can do
  - Restrictions similar to those of timer:
    - Can't access user-space

    - Can't sleep or do anything that may sleep, including mem allocation or grabbing semaphores.
    - Can't call the scheduler
- Handler arugments and return value:
    - int **request_irq**(unsigned int irq, irqreturn_t (*handler) (int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);
    - 1: "irq", the irq number
    - 2: "*dev_id", private data (same as passed to request_irq(), can pass internal device "instance" pointer to easily find instance on interrupt).
    - 3: "*regs", the CPU registers at interrupt occurence, seldom used.

- retval is status of interrupt handling:
  - IRQ_HANDLED, an interrupt occured. Should also be used in no way to determine if interrupt did occur.
  - IRQ_NONE, no interrupt occurred. Interrupt was suprious or is shared.
  - Macro for generating return value depending on variable (non-zero means interrupt handled):
    - IRQ_RETVAL(var);
- Enabling and disabling interrupts:
  - Try to avoid in as much as possible
  - No way to disable interrupts on all processors in the same time.

- Disabling a single interrupt:
  - <asm/irq.h>
  - void **disable_irq**(int irq);
    - Wait for interrupt handler if it's running and disable it
    - Careful with deadlocks
  - void **disable_irq_nosync**(int irq);
    - Disables interrupt without checking if handler is running
  - void **enable_irq**(int irq);
  - May play wiht PIC's mask
  - Calls can be nested
- Disabling all interrupts:
  - Disable interrupts on local CPU
  - <asm/system.h>

- void **local_irq_save**(unsigned long flags);
  - Disables interrupts and saves local interrupt flags
- void **local_irq_disable**(void);
  - Disables without recording flags
- void **local_irq_restore**(unsigned long flags);
  - Restores interrupt flags
- void l**ocal_irq_enable**(void);
  - Reenables local interrupts
- No nesting possible: use local_irq_save().

3. Top and bottom halves

- Interrupt handlers must finish rapidely (top half)
- Lengthy work defered to later in "bottom half" with all interrupts enabled.
- Usually:
  - Top half records device data to temporary buffer and scheds BH. Network code pushed packet up stack, while actual processing done in BH.
  - BH further does whatever scheduling is needed
- Tasklets: Overview
  - Will be covered in detail in Ch.8
  - Reminder: from “Using Linux in Embedded Systems”:

- Runs in software interrupt context
- Only one tasklet of a given type will ever be running in the same time in the entire system.
- Even if rescheded multiple times, will only run once.
- Interrupt may occur while tasklet is running => use appropriate locks.
- Tasklets run on the same CPU where they scheduled

- API reminder:
  - DECLARE_TASKLET()
  - tasklet_init()
  - tasklet_schedule()

- Workqueues:
  - Issue function within work queue process context

- Reminder:
  - Can sleep
  - Can't access user-space
  - Can use system default work queue

4. Interrupt sharing

- Interrupt sharing a must on modern hardware (used by PCI).
- Installing a shared handler:
  - Use same request_irq()
  - Difference with non-shared handlers:
    - Use the SA_SHIRQ flag
    - Pass non-NULL, unique ID in *dev_id, kernel complains otherwise as it may oops at irq freeing.
  - Registration fails if other handlers have registered without setting the SA_SHIRQ flag.

- On interrupt occurence, kernel invokes every registered handler for the given interrupt, passing it the *dev_id provided on registration.
- Each handler must determine if the device it handles issued the interrupt, and return IRQ_NONE otherwise.
- *dev_id's importance is shown when free_irq() is called since that's the only way to know **which** handler should be removed from shared list.

- Running the handler
  - Remember that interrupt is shared => don't disable interrupts.
- The /proc interface and shared interrupts
  - /proc/interrupts shows list of driver sharing a given interrupt.

5. Interrupt-driven I/O

- Reminder: use buffer I/O
- Buffered I/O allows for interrupt-driven I/O
- Usually, best if hardware:
  - Generates int when data is ready for read
  - Generates int when more data can be written or generate int to ack data writing.
- Guidelines:
  - Keep in mind that your driver may "miss" interrupts. For that reason, it's always a good idea, when appropriate, to set up a timer function to check whether interrupts have been missed.

Timely execution and time measurement

1. Measuring time lapses
2. Knowing the current time
3. Delaying execution
4. Kernel timers
5. Tasklets
6. Workqueues

1. Measuring time lapses

- Background:
  - Kernel timer: the kernel's heartbeat
  - Interval configured at boot time
  - HZ in <linux/param.h>
  - Value ranges according to architecture
  - For x86, PPC: 1000; For MIPS: 100
  - Can change HZ if needed
  - For each timer interrupt: kernel increments internal counter
    - jiffies_64 (even on 32-bit platforms)
    - Access to jiffies_64 not atomic on 32-bit platforms

- Drivers typically use jiffies (unsigned long):
  - Same as jiffies_64 or least significant bits of jiffies_64.
- Using the jiffies counter:
  - jiffies and jiffies_64: <linux/sched.h>
  - No need to hold any locks to read jiffies
  - jiffies is volatile: will be fetched from RAM on every read.
  - Never write jiffies
  - jiffies may wrap-around, comparisons should be properly done.

- Comparison functions:
  - <linux/jiffies.h>
  - int **time_after**(unsigned long a, unsigned long b);
    - True if "a" is after "b"
  - int **time_before**(unsigned long a, unsigned long b);
    - True if "a" if before "b"
  - int **time_after_eq**(unsigned long a, unsigned long b);
    - After or equal
  - int **time_before_eq**(unsigned long a, unsigned long b);
    - Before or equal
- Obtaining time difference:
  - diff = (long) t2 - (long) t1

- Converting time to milliseconds:
  - msec = diff * 1000/HZ;
- User-space uses "struct timeval" and "struct timespec"
- Converting jiffies to/from "struct timeval" and "struct timespec"
  - <linux/time.h>
  - unsigned long **timespec_to_jiffies**(struct timespec *value);
  - void **jiffies_to_timespec**(unsigned long jiffies, struct timespec *value);
  - unsigned long **timeval_to_jiffies**(struct timeval *value);

    - void **jiffies_to_timeval**(unsigned long jiffies, struct timeval *value);
  - Helper function in case you need to read jiffies_64:
    - <linux/jiffies.h>
    - u64 **get_jiffies_64**(void)
    - Typically no need to read jiffies_64 on 32-bit systems
  - In user-space, HZ is always seen as 100. Helper functions maintain view.
- Processor-specific registers:
  - Some processors have internals counters counting the number of CPU clock cycles.
  - Very useful for precise time-measurements

- Very architecture specific: size, user-space visibility, read-only.
- On x86, TSC (64 bit)
- On MIPS, register 9 of "coprocessor 0" (VR4181): 32-bit
- There exist platform-specific functions, but there's also a generic helper:
  - <linux/timex.h>
  - cycles_t **get_cycles**(void);
  - Returns 0 on processors without CPU counters
  - cycles_t is unsigned type

- x86-type assembly macros:

  rdtsc(low32,high32);

  rdtscl(low32);

  rdtscll(var64);
- Often not syncrhonized on SMP systems

2. Knowing the current time

- Time representation tied to jiffies
- Usually drivers need just jiffies
- In some rare cases, drivers need to know exact real-world time.
- Converting wall-time (user-space time) to jiffies:
  - <linux/time.h>
  - unsigned long **mktime**(unsigned int year, unsigned int mon, unsigned int day, unsigned int hour, unsigned int min, unsigned int sec);
- Getting absolute timestamp:

- <linux/time.h>
- void **do_gettimeofday**(struct timeval *tv);
- Microsecond resolution

- Getting current time:
  - <linux/time.h>
  - struct timespec **current_kernel_time**(void);
  - jiffy resolution

3. Delaying execution

- Long delays
  - Busy waiting
    - Wait in tight loop for a certain time:

```
while (time_before(jiffies, deadline))
        cpu_relax();
```

    - The cpu_relax() call doesn't usually do much
    - This technique is discouraged
    - Shouldn't be done while interrupts disabled
    - May end up waiting much more than expected because of scheduling.
  - Yielding the processor
    - Call the scheduler:

```
while (time_before(jiffies, deadline)) schedule();
```

- May result in process looping rapidly if only process on CPU.
- May end up waiting a very long time if process is loaded

- Timeouts
  - Some types of wait can be timed-out:
    - <linux/wait.h>
    - long **wait_event_timeout**(wait_queue_head_t q, condition, long timeout);
    - long **wait_event_interruptible_timeout**(wait_queue_head_t q, condition, long timeout);
    - timeout is number of jiffies, not absolute time
    - retval is zero if timeout
    - retval is remaining delay if woken up
  - If no event is waited for:
    - <linux/sched.h>

- signed long **schedule_timeout**(signed long timeout);
- timeout is number of jiffies
- Must set current process to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE.
- retval is zero unless return before timeout (because of signal)
- After call, process is set to TASK_RUNNING

- Short delays:
  - Sometimes short busy-waits needed for hardware ops
  - Helper functions:
    - <linux/delay.h>
    - void **ndelay**(unsigned long nsecs); /* nanoseconds */
    - void **udelay**(unsigned long usecs); /* microseconds */
    - void **mdelay**(unsigned long msecs); /* milliseconds */

- Will typically wake up **after** delay is expired
- ndelay() and udelay() have upper-limit: if static value passed is too large, module will fail to load with __bad_udelay.
- Non-busy wait functions:
  - <linux/delay.h>
  - void **msleep**(unsigned int millisecs);
    - Uninterruptible sleep
  - unsigned long **msleep_interruptible**(unsigned int millisecs);
    - Interruptible sleep
    - retval zero if delay achieved
    - retval is number of milliseconds early if woken before timeout

- void **ssleep**(unsigned int seconds);
  - Uninterruptible sleep
- Likely wake up much later than delay

4. Kernel timers

- Background
  - Notified in due time without blocking
  - Schedule callback execution
  - Jiffies resolution
  - Asynchronous execution of callbacks
  - Registration:
    - Provide callback function
    - Provide delay
    - Provide parameter to callback function
  - Limitations:
    - Use software interrupt locks (_bh())

- No "current process" context
- Must not access user-space
- "current" has no relation to callback
- Cannot sleep or reschedule anything
- Soft-real-time, not hard-real-time

- Testing current context:
  - <asm/hardirq.h>
  - **in_interrupt**()
  - **in_atomic**() /* Cannot schedule*/
- A timer callback can reschedule itself
- On SMP, callback runs on same CPU as registered

- Timers can cause race-conditions: use appropriate locks

- The timer API:
  - <linux/timer.h>

```
struct timer_list {
    ...
    unsigned long expires;              /* jiffies */
    void (*function)(unsigned long);
    unsigned long data;
}
```

  - Static initialization:
    - struct timer_list **TIMER_INITIALIZER**(_function, _expires, _data);

- Dynamic initialization:
  - void **init_timer**(struct timer_list *timer);
- May change the 3 fields in struct after initialization
- Adding timer to list:
  - void **add_timer**(struct timer_list *timer);
- Deleting timer prior to expiry:
  - void **del_timer**(struct timer_list *timer);
- Modify timer expiry:
  - int **mod_timer**(struct timer_list *timer, unsigned long expires);
- Delete timer and, on return, make sure it's not running on any CPU:

    - int **del_timer_sync**(struct timer_list *timer);
    - May sleep if not in atomic context
    - Careful when calling while holding locks => deadlock if timer function attempts to obtain same lock
  - Indicate if timer is currently scheduled for execution:
    - int **timer_pending**(const struct timer_list *timer);
- The implementation of kernel timers:
  - Per-cpu data structure
  - Timers inserted into per-cpu struct using internal_add_timer()

- Timers inserted in "cascading table" depending on expiry:
  - negative expiry time: scheduled to run at next ticik
  - between 0 to 255 jiffies (bits 1-8 in "expires": 256 lists
  - between 255 and 16,384 jffies (bits 9-14 in "expires"): 64 lists.
  - bits 15-20: 64 lists
  - bits 21-26: 64 lists
  - bits 27-31: 64 lists
  - Larger values: hash
- When __run_timers is executed:
  - Executes all timers for current tick

- If jiffies is mutltiple of 256, rehash next level into 256 lists, and cascade other levels as needed.

5. Tasklets

- Somewhat similar to timers:
  - Run in soft interrupt context
  - Run on same CPU where registered
  - Received unsigned long argument provided on registration.
  - Can reregister themselves
  - Tasklet lists are per-cpu
- Difference from timers:
  - No specific time for execution
  - Just pending work for a later time

- Especially useful for interrupt handlers to delay work for "later".
- The API:
  - <linux/interrupt.h>

```
struct tasklet_struct {
...
void (*func)(unsigned long);
unsigned long data;
}
```

- Static initialization:
  - DECLARE_TASKLET(name, func, data);
  - DECLARE_TASKLET_DISABLED(name, func, data);

- Dynamic initialization:
  - volid **tasklet_init**(struct tasklet_struct *t, void (*func) (unsigned long), unsigned long data);
- Tasklet features:
  - Can disable/enable, even multiple times
  - Will only run if enabled as often as it was disabled
  - Can specify "high" or "low" priority tasklets, the former being executed first.
  - Will run either immediately, if no system load, or at the next system tick at the latest.
  - Many tasklets can run in parallel on many CPUs

- Only one tasklet of a given type can run on any CPU at a given time.

- Who runs tasklets?
  - Per-cpu ksoftirqd
- Full API:
  - void **tasklet_disable**(struct tasklet_struct *t);
    - Disable tasklet execution
    - Busy wait if tasklet currently running
  - void **tasklet_disable_sync**(struct tasklet_struct *t);
    - Disable tasklet execution
    - Don't wait for running tasklet to finish

- void **tasklet_enable**(struct tasklet_struct *t);
  - Enable tasklet execution
- void **tasklet_schedule**(struct tasklet_struct *t);
  - Schedule for execution
  - If resched, run once
  - If resched while running, rerun
- void **tasklet_hi_schedule**(struct tasklet_struct *t);
  - Schedule with high priority
  - Avoid unless absolutely necessary (ex.: media streaming)
- void **tasklet_kill**(struct tasklet_struct *t);
  - Make sure tasklet doesn't run again
  - Usually on device close

- Will block if tasklet scheduled
- If tasklet reschedules itself, must make sure it doesn't prior to using tasklet_kill().

6. Workqueues

- Basics:
  - Not the same thing as previously seen wait queues
  - Similar to tasklets:
    - Run something in the future
    - Usually runs on same processor registered
    - Cannnot access userspace
  - Different from tasklets:
    - Run in special process context, not software interrupt context.
    - Can sleep
    - Can ask for delayed execution for specific time
    - No need for atomic execution

- Can tolerate high latency
- Each work queue has dedicated "kernel thread"

- Basic API:
  - struct workqueue_struct: <linux/workqueue.h>
  - struct wrokqueue_struct ***create_workqueue**(const char *name);
    - One kernel thread per cpu
  - struct workqueue_struct ***create_singlethread_workqueue**(const char *name);
    - One kernel thread for entire system
- Submitting a task to a work queue:
  - Static declaration:
    - DECLARE_WORK(name, void (*function)(void *), void *data);

- Dynamic declaration:
  - **INIT_WORK**(struct work_sutrct *work, void (*function)(void *), void *data);
  - **PREPARE_WORK**(struct work_struct *work, void (*function)(void *), void *data);
    - Similar to INIT_WORK() but doesn't initialize pointers to link struct work_struct to actual work queue.
    - Useful if structure may have already been submitted to work queue.
- Actual submission:
  - int **queue_work**(struct workqueue_struct *queue, struct work_struct *work);
    - retval is zero if successfull add
    - retval nonzero if already in queue (not added again)
  - int **queue_delayed_work**(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);

- About work queue callback sleep:
  - Will affect other callbacks queued in work queue.
- Rest of API:
  - int **cancel_delayed_work**(struct work_struct *work);
    - retval nonzero if cancel prior to execution
    - retval zero if work may have been running (and could still be running on different CPU.
  - void **flush_workqueue**(struct workqueue_struct *queue);
    - Make sure no scheduled work is running anywhere in the system.
  - void **destroy_workqueue**(struct workqueue_struct *queue);
    - Free work queue.

- The share queue:
  - Not all drivers need their own work queue
  - Kernel provides default shared queue
  - Make sure your task doesn't monopolize queue
  - API:
    - int **schedule_work**(struct work_struct *work);
    - int **schedule_delayed_work**(struct work_struct *work unsigned long delay);
    - void **flush_scheduled_work**(void);
  - Can still use cancel_delayed_work().

Memory resources

1. The real story of kmalloc
2. Lookaside caches
3. get_free_page and friends
4. The alloc_pages interface
5. vmalloc and friends
6. Per-CPU variables
7. Obtaining large buffers
8. Memory management in Linux
9. The mmap device operation

10. Performing direct I/O
11. Asynchronous I/O
12. Direct memory access
13. The generic DMA layer

1. The real story of kmalloc

- Easy to use
- Equivalent to malloc()
- Allocated region is physically contiguous
- Must flush content of kmallc'ed regions if shared with uspace for security.
- Prototypes:
  - <linux/slab.h>
  - void ***kmalloc**(size_t size, int flags);
  - void **kfree**(const void *ptr);

- The flags argument:
  - Flags defined in <linux/gfp.h>
  - Flags prefix is GFP_ because kmalloc typically relies on an internal function known as __get_free_pages().
  - Basic flags:
    - GFP_ATOMIC:
      - Never sleep to get memory
      - Useful in interrupt handlers, tasklets and timers
      - May fail if emergency reserve is full (a few free pages)
    - GFP_KERNEL:
      - Normal kernel allocation
      - May sleep

    - Used by functions servicing system calls on behalf of process
    - Caller must be reentrant
    - Caller must not be holding locks
- GFP_USER:
    - Allocate for user-space
    - May sleep
- GFP_HIGHUSER:
    - Allocates high-memory (if available)
    - May sleep
- GFP_NOIO:
    - Similar to GFP_KERNEL
    - Indicates to kernel not to do any I/O to satisfy request
- GFP_NOFS:
    - Similar to GFP_KERNEL
    - Indicates to kernel not to do any filesystem calls

- Additional flags to OR "|" with basic flags to further detail alloc:
  - __GFP_DMA:
    - Allocate region that can be used for DMA
    - Architecture specific
  - __GFP_HIGHMEM:
    - Allocate high mem if possible
  - __GFP_COLD:
    - Allocate memory outside the processors' cache
    - Useful for allocating memory for DMA reads
  - __GFP_NOWARN:
    - Don't printk kernel warning if allocation fails
  - __GFP_HIGH:
    - Very high priority request

        - Tries to use all of the kernel's last reserves
    - __GFP_REPEAT:
        - Modify allocator behavior to repeat if fail
        - May still fail
    - __GFP_NOFAIL:
        - Tell allocator never to fail
        - Should never be used in driver
    - __GFP_NORETRY:
        - Tell allocator to fail immediately if region not available
- Memory zones:
    - Minimum zones recognized on all platforms by Linux:
        - Normal memory:
            - Region where allocations typically occur
        - DMA memory:
            - Memory "preferred" by architecture for DMA

- High memory:
  - For very large allocation
  - Typically requires special structures to be set up to access the high memory.

- The size argument:
  - Space not managed like user-space malloc() (using a heap).
  - Page-oriented allocation
  - Pools of memory objects
  - Actual implementation details complicated
  - Possible allocation sizes pre-defined, likely will get more than what is requested.

- Smallest unit: 32 or 64 bytes (depending on size of pages in architecture).
- Largest unit: 128k

2. Lookaside caches

- Custom functionality for drivers allocating same-sized objects repeatidly.
- <linux/slab.h>
- kmem_cache_t ***kmem_cache_create**(const char *name, size_t size, size_t offset, unsigned long flags, void (*constructor)(void *, kmem_cache_t *, unsigned long flags), void (*destructor)(void *, kmem_cache_t *, unsigned long flags),
  - "name" is for housekeeping: no blanks, must be static string.
  - Creates pool of objects with "size" size

- "offset" of object in page (if alignment needed). Usually "0".
- "flags" bitmask:
  - SLAB_NO_REAP:
    - Do not resize cache when memory allocator is looking for memory.
    - Discouraged
  - SLAB_HWCACHE_ALIGN:
    - Align with CPU cache lines
    - May be good for performance on SMP machines
    - May waste lots of memory
  - SLAB_CACHE_DMA:
    - Allocate in DMA region

- "constructor"/"destructor"
  - Optional
  - Initialize newly allocated objects / cleanup objects prior to free.
  - Constructor called after allocation, not necessarily immediately.
  - Destructors may be called at any time after free request
  - May or may not sleep depending if "flags" passed contains.
  - SLAB_CTOR_ATOMIC
  - Can use same function for both
  - Actual constructor called with SLAB_CTOR_CONSTRUCTOR

- void ***kmem_cache_alloc**(kmem_cache_t *cache, int flags);
  - Allocate memory from cache
  - Flags (same as kmalloc) used in case cache needs to allocate more mem.
- void **kmem_cache_free**(kmem_cache_t *cache, const void *obj);
  - Free cache-allocated memory
- int **kmem_cache_destroy**(kmem_cache_t *cache);
  - Destroy entire cache (usually on module_exit)

  - Fails if not all objects have been kmem_cache_free'd
  - Check retval for failure (mem leak)
- Statistics on cache usage available from /proc/slabinfo.
- Memory pools:
  - For places where memory allocation can't fail
  - Like a lookaside cache with reserves for emergencies.
  - Built on top of lookaside cache functionality
  - Typically should not be used in drivers
  - <linux/mempool.h>

- mempool_t * **mempool_create**(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void *pool_data);
  - "min_nr": minimum number of objects to keep available
  - typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
    - Usually set to "mempool_alloc_slab"
  - typedef void (mempool_free_t)(void *element, void *pool_data);
    - Usually set to "mempool_free_slab"
  - "pool_data" is passed to alloc_fn and free_fn
  - mempool_alloc_slab and mempool_free_slab rely on kmem_cache_alloc and kmem_cache_free.

- Typically:

```
cache = kmem_cache_create(...);
pool = mempool_create(MY_POOL_MINIMUM,
mempool_alloc_slab,
mempool_free_slab,
cache);
```

- void ***mempool_alloc***(mempool_t *pool, int gfp_mask);
  - Allocate from pool.
- void **mempool_free**(void *element, mempool_t *pool);
  - Free from pool.
- int **mempool_resize**(mempool_t *pool, int new_min_nr, int gfp_mask);

- Resize memory pool
- Check retval for success

- void **mempool_destroy**(mempool_t *pool);
  - Must free all objects prior to mempool destruction
  - Kernel oops otherwise

3. get_free_page and friends

- Necessary for allocating large chunks of memory
- Per-page allocation will result in less memory waste than kmalloc().
- <linux/slab.h>
- Page allocation:
  - **get_zeroed_page**(unsigned int flags);
    - Get pointer to new page preinitialized with zeroes
  - **__get_free_page**(unsigned int flags);
    - Get pointer to new page without clearing content
  - **__get_free_pages**(unsigned int flags, unsigned int order);

- Get 2^order physically contiguous pages without clearing content.
- Fails if order too big
- Can use get_order() to get order from int value
- Maximum allowed order is 10 or 11 (depending on arch)
- The further from reboot, the less the chances a large alloc will succeed.
- For information on what orders of allocation are available:

  /proc/buddyinfo

- "flags" same as for kmalloc

- Page freeing:
  - void **free_page**(unsigned long addr);

- void **free_pages**(unsigned long addr, unsigned long order);
  - Different "order" from allocation will cause memory corruption.
- Page allocation can be done anywhere, pending the same restrictions as kmalloc().
- Requesting too much memory can result in system performance degradation.

4. The alloc_pages interface

- Should be used when using nitty-gritty functionality requiring access to struct page, the kernel's internal description of a memory page.
- <linux/slab.h>
- struct page ***alloc_pages_node**(int nid, unsigned int flags, unsigned int order);
  - "nid" is NUMA ID, not typically used by most drivers.
  - "flags" same as for kmalloc()
  - "order" same as for __get_free_pages()

- struct page *__alloc_pages__(unsigned int flags, unsigned int order);
  - Same as alloc_pages_order(), but for local NUMA node
- struct page *__alloc_page__(unsigned int flags);
  - Simplest allocation
- void ___free_page__(struct page *page);
  - Free single page
- void ___free_pages__(struct page *page, unsigned in order);
  - Free page lot

- void **free_hot_page**(struct *page);
  - Free single page that is in cache
- void **free_cold_page**(struct *page);
  - Free single page that is not in cache

5. vmalloc and friends

- Contiguous address range in virtual memory
- May be physically discoutiguous
- Discouraged in most situations:
  - Memory allocated by vmalloc() slightly less efficient
  - Amount of memory set asside for vmalloc() limited on some archs.
  - Must conveniently set up entire page table entries for obtaining a contiguous virtual address space.
  - Can't be used for DMA

  - Can't be used in an atomic context (relies on GFP_KERNEL).
- Appropriate for large software-only buffers
- Used by kernel's module functionality to allocate space for modules.
- Used by relayfs
- <linux/vmalloc.h>
- void ***vmalloc**(unsigned long size);
  - retval is zero on failure
  - retval is pointer to region of at least "size" size

- void **vfree**(void *addr);
  - Free vmalloc'ed memory

6. Per-CPU variables

- New to 2.6.x
- Creating a per-cpu variable creates one copy for each CPU.
- No contention on variable
- Good caching
- Heavily used by networking subsystem
- Some ports have limited memory available for per-CPU variables:
  - Use wisely.
- <linux/percpu.h>

- Static declaration:
  - DEFINE_PER_CPU(type, name);
    - Type can be array (char[10])
- Dynamic declaration:
  - void ***alloc_percpu**(type);
  - void ***__alloc_percpu**(size_t size, size_t align);
    - Use in case of special alignment needs
  - **free_percpu**();
- Can be manipulated without locks, pending preemption protection.

- For statically-allocated variables:
  - **get_cpu_var**(var);
    - Get local CPU's value
    - Disable preemption
  - **put_cpu_var**(var);
    - Enable preemption
  - **per_cpu**(var, cpu_id);
    - Get another CPU's variable
    - Must use locking

- For dynamically-allocated variables:
  - int **get_cpu**(void);
    - Get current CPU's ID and disable preemption
  - **per_cpu_ptr**(void *per_cpu_ver, int cpu_id);
    - Access variable on a given CPU
  - void **put_cpu**(void);
    - Enable preemption
  - get_cpu() and put_cpu() should only be needed when accessing the local processor's dynamically-allocated variable.

- May export per-cpu variables:
  - **EXPORT_PER_CPU_SYMBOL(per_cpu_var);**
  - **EXPORT_PER_CPU_SYMBOL_GPL**(per_cpu_var);
- To use from another module:
  - **DECLARE_PER_CPU**(type, name);
    - In contrast with DEFINE_PER_CPU()
- Canned per-cpu counters:
  - <linux/percpu_counter.h>

7. Obtaining large buffers

- Large allocation are prone to failure
- The more the system runs, the more its memory gets fragmented.
- Acquiring a dedicated buffer at boot time:
  - Best chances of getting large buffers is at boot time
  - Considered "dirty" trick
  - **Only drivers linked in kernel can allocate boot-time memory.**
  - Modules can try by getting loaded very early after boot and, therefore, calling __get_free_pages() prior to any memory fragmentation.

- <linux/bootmem.h>
- void ***alloc_bootmem**(unsigned long size);
  - Allocate non-page-aligned memory
  - May allocate high memory
- void ***alloc_bootmem_low**(unsigned long size);
  - Allocate non-page-aligned memory
  - Allocate low memory
  - Typically for DMA
- void ***alloc_bootmem_pages**(unsigned long size);
  - Allocate page-aligned memory
  - May allocate high memory

- void ***alloc_bootmem_low_pages**(unsigned long size);
  - Allocate page-aligned memory
  - Allocate low memory
  - Typically for DMA
- void **free_bootmem**(unsigned long addr, unsigned long size);
  - In the unlikely case where you'd like to free this memory
  - Having called this, further attempts to reallocate the same area will most likely fail.

8. Memory management in Linux

- Address types:
  - See Fig 15-1 on p.414
  - User virtual address:
    - Address within user-space process
  - Physical address:
    - Actual address put by CPU on system bus
    - If low-mem, use __va() to convert to virtual address
  - Bus address:
    - Address for accessing peripherals
    - Often mapped to physical address, but not always

- Kernel logical address:
  - Virtual addresses that map directly to physical addresses by an offset.
  - kmalloc() hands out this type of memory
  - Use __pa() to convert to physical address
- Kernel virtual address:
  - Virtual address that doesn't necessarily map to a given range of physical address.
  - A continguous range of kernel virtual addresses will typically not be physically contiguous. It will be contiguous in virtual space because of the page mappings.
  - vmalloc()'ed memory

- Physical addresses and pages:
  - Memory subdivided in pages, usuall 4K (PAGE_SIZE: <asm/page.h>.
  - Address is always aggregate: <page frame number><offset in page>.
  - PAGE_SHIFT is number of bits in page offset
- High and low memory:
  - Hardware limitation:
    - 32-bit system can only address 4GB of physical memory
  - Linux-specific limitations:

- Virtual address space is usually split between kernel and process:
  - 3GB for process / 1GB for kernel
- Kernel can't handle memory not mapped in its space (logical address):
  - Therefore, for a long time, 1GB RAM was all Linux supported
- Modern processors:
  - Have been modified so that they can handle >4GB RAM
  - Only portion of the RAM has logical addresses (~1GB)
- Linux's use of these extensions:
  - Since kernel needs to run in logical address space, kernel is located < 1GB RAM.
  - Memory beyond that is used for user-space processes

- Most embedded systems have no high memory to worry about.

- The memory map and struct page:
  - Any reference to physical memory handled by ref to struct page.
  - struct page: <linux/mm.h>
    - atomic_t count;
      - Number of refs to page
      - Deallocation on refcount 0
    - void *virtual;
      - If mapped (usually is for low-mem), page's virtual address
      - If not mapped (probably high-mem), NULL
      - Not present on all archs

- unsigned long flags;
  - Page use flags
- Address conversion and struct page:
  - struct page ***virt_to_page**(void *kaddr);
    - <asm/page.h>
    - Get struct page using kernel logical address
    - Not for kernel virtual addresses
  - struct page ***pfn_to_page**(int pfn);
    - Get struct page from Page Frame Number
  - void ***page_address**(struct page *page);
    - <linux/mm.h>
    - Get virtual address using struct page
- Mapping struct page:

- Simple mapping:
  - <linux/highmem.h>
  - void ***kmap**(struct page *page);
    - If low-mem, return logical address
    - If high-mem, maps high-mem to kernel space
    - May sleep to create mapping
  - void **kunmap**(struct page *page);
    - Freeing mapping done with kmap
- Atomic mapping:
  - <linux/highmem.h>
  - <asm/kmap_types.h>
  - void ***kmap_atomic**(struct page *page, enum km_type type);
    - Atomic form of kmap()
  - void **kunmap_atomic**(void *addr, enum km_type type);
    - Freeing mapping done with kmap_atomic

- Page tables:
  - Mapping from virtual address to physical page
  - Built and maitained by OS
  - Used by CPU to convert addresses
  - No need for direct page table manipulation in drivers
- Virtual memory areas:
  - Basics:
    - "Memory object with its own properties":
      - Virtually contiguous region
      - Set of permission flags
      - Same object

- Processes usually have following areas:
  - Text section (.text)
  - Initialized data (.data)
  - Unitialized data (.bss)
  - Stack
  - Memory mappings
- See /proc/<pid>/maps for example layout
- Each line in maps has format:

  start-end perm offset major:minor inode image

- The vm_are_struct structure:
  - Definition of a VMA
  - <linux/mm.h>
  - unsigned long vm_start; unsigned long vm_end;
    - Virtual address range for VMA

- struct file *vm_file;
  - File associated with VMA if memmap
  - May be NULL
- unsigned long vm_pgoff;
  - Offset in pages within mapped file if memmap
- unsigned long vm_flags;
  - Description of region
  - VM_IO: do not include in process core dump
  - VM_RESERVED: do not swap (typical for driver)
- struct vm_operations_struct *vm_ops;
  - Ops to work on VMA
- void *vm_private_data;
  - Private data

- The vm_operations_struct structure:
  - void (*open)(struct vm_area_struct *vma);
    - Called every time new ref to area (like fork)
  - void (*close)(struct vm_area_struct *vma);
    - Called every time ref to area is closed (like close())
  - struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);
    - Access to page but page isn't there
    - If NULL, empty page is allocated by kernel
  - int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock);
    - Allows VMA to be initialized prior to being referenced
    - Not needed by drivers

- The process memory map:
  - Each process has struct mm_struct (linux/sched.h)
  - struct mm_struct contains description for process's struct.
  - current->mm
  - Can be shared amongst threads

9. The mmap device operation

- Basics:
  - Ability to map file or device directly to process' address space.
  - X maps /dev/mem
  - PCI generally lends itself well to mmaping
  - Read/writing to area results in direct read/write to device.
  - Mapped areas must be page-aligned
  - Mapped areas must have sizes which are a multiple of PAGE_SIZE.

- System call goes through lots of processing before driver's mmap function gets called.
- From user-space:
  - void ***mmap**(caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
- Actualy driver callback prototype:
  - int (***mmap**)(struct file *filp, struct vm_area_struct *vma);
  - "vma" is already initialized prior to callback issued
- Driver must:
  - Populate page tables
  - Replaced vma-ops if necessary

- Populating page tables:
  - remap_pfn_range
  - nopage
- Using remap_pfn_range:
  - To be used by driver's mmap() callback
  - int **remap_pfn_range**(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn, unsigned long size, pgprot_t prot);
    - pfn is actual RAM
  - int **io_remap_pfn_range**(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long phys_addr, unsigned long size, pgprot_t prot);

    - phys_addr is I/O memory
- On most archs, io_remap_pfn_range == remap_pfn_range.
- Many drivers directly use remap_pfn_range instead of io_remap_pfn_range.
- Fields:
    - vma:
        - VMA where range is to be mapped
    - virt_addr:
        - User region where to start mmapping
    - pfn:
        - The physical PFN to which region should be mapped
        - Physical address right-shifted by PAGE_SHIFT

    - Typically vma->vm_pgoff contains this value
  - size:
    - Size of region to be mmaped in bytes
  - prot:
    - Protection bits
    - Use vma->vm_page_prot
- Must test retval for success (0 mean fail)
- About caching:;
  - As stated earlier, I/O mem should not be cached
  - Should be taken care of by BIOS/bootloader/early-boot-code.
  - Can be set via protection field in VMA
  - See drivers/char/mem.c:pgprot_nocached() for example

- A simple implementation:

```
static int my_driver_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end -
      vma->vm_start, vma->vm_page_prot))
          return -EAGAIN;
    return 0;
}
```

- Adding VMA operations:
  - If driver needs to do further processing every time a process gets a reference to memmaped area (like on a fork) or when a process doesn't refer to region anymore (like on close()).

- Modify vm_ops prior to returning from mmap() callback:

```
vma->vm_ops = &my_vma_ops;
```

  - Most drivers don't need to do this
- Mapping memory with nopage:
  - Sometimg remap_pfn_range() not flexible enough
  - If using nomap, actual mmap() callback just sets a few fields in the vma and changes the vm_ops, but doesn't actually call on remap_pfn_range().
  - nopage is necessary for mremap() syscall

- As shown before:
  - struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);
- nopage invoked on access to address not mapped
- If process does mremap and driver doesn't provide nopage, a new page filled with zero is provided to process.
- Can define simple nopage function that just returns NOPAGE_SIGBUS if you don't want any user-space process to be able to do a mremap.
- See LDD3 for full details

- Remapping specific I/O regions:
  - Can check in mmap() callback to verify if uspace process is trying to mmap more I/O memory than available on device.
  - Return -EINVAL if too large
- Remapping RAM:
  - Can only use remap_pfn_range on:
    - Physical ranges above RAM
    - RAM-locked pages (reserved pages) / can't be swapped
  - Attemtps to remap virtual addresses will result in mapping of the "zero page".

- Remapping RAM with the nopage method:
  - See LDD3 for example of how to map physical pages allocated in logical addresses to uspace.
- Remapping Kernel virtual addresses:
  - See LDD3 for how to map virtual address ranges to uspace.
- Relayfs:
  - If you ever need to share buffers between kernel space and user space, use relayfs.
  - Takes care of allocation of large buffers
  - Allows mmap() from user-space
  - Provides efficient and atomic logging functions to fill buffer.

10. Performing direct I/O

- Allow direct reading/writing using a user-space buffer without ever copying the data to kernel-space first.
- Very useful for stuff like block and networking devices.
- However, most drivers that need direct I/O already have subsystems that do the dirty work, such as in the case of block and networking.
- Kernel provides get_user_pages() to lock user pages in RAM for transfer.

- Once done, must mark pages as dirty if modified:
  - void **SetPageDirty**(struct page *page);
- Remove from page cache:
  - void **page_cache_release**(struct page *page);
- See LDD3 for full details

11. Asynchronous I/O

- Initiate I/O without waiting for actual transfer to occur.
- Network and block drivers asynchronous by default.
- Async API valid for char drivers only
- Example drivers worth implementing async I/O for:
  - Streaming
- Typically involves carrying out direct I/O

- Relevant char dev callbacks:
  - aio_read()
  - aio_write()
  - aio_fsync()
- See LDD3 for full details

12. Direct memory access

- Overview of a DMA data transfer:
  - Basically:
    - 1- Set up hardware for transfer
    - 2- Respond to interrupt by signaling that transfer is done
  - Usually on read:
    - 1- Process read
    - 2- Driver sets up hardware to transfer from device to DMA buffer.
    - 3- Driver puts process to sleep
    - 4- Hardware writes data to buffer
    - 5- Hardware issues interrupt
    - 6- Interrupt handler deals with transfered data and awakens process.

- Usually on write:
  - 1- Process write
  - 2- Data copied to DMA buffer
  - 3- Driver sets up hardware to transfer from DMA buffer to device.
  - 4- Driver puts process to sleep
  - 5- Hardware copies data from buffer to device
  - 6- Hardware issues interrupt
  - 7- Interrupt handler wakes up process
- For asynchronous data arrival (like data acquisition):
  - 1- Interrupt signals arrival of new data
  - 2- Interrupt handler instructs hardware to transfer data to designated buffer.

    - 3- Hardware writes data to buffer
    - 4- Hardware raises interrupt to notify of end of transfer
    - 5- Interrupt handler dispatches data
  - Network cards operate using DMA ring buffer shared with CPU.
- Allocating the DMA buffer:
  - DMA buffers must be physically contiguous
  - Devices know nothing of virtual mapppings
  - Sometimes there are limitations even on the region where a DMA buffer may be allocated from.
  - Use of GFP_DMA

- Do-it-yourself allocation:
  - Use kmalloc() or get_free_pages() presented earlier
  - May have problems if buffer required is too large
  - Can also reserve RAM at boot time by telling kernel not to use all RAM:
    - Use ioremap() later to map region for I/O
    - Doesn't work on systems with high-mem
  - Can use scatter/gather I/O if device allows it
- Bus addresses:
  - Devices only recognize physical addresses
  - On some platforms, devices use "bus addresses", which are not the same as "physical addresses". (I/O addresses mapped onto bus ...)

- Conversion functions exist, but strongly discouraged (deprecated):
  - unsigned long **virt_to_bus**(volatile void *address);
  - void ***bus_to_virt**(unsigned long address);
- Use generic DMA layer instead for obtaining the proper addresses.

- DMA for ISA devices:
  - See LDD3

13. The generic DMA layer

- Basics:
  - Kernel provides DMA layer to take care of all the possible complexities of DMA operation accross all platforms in a portable fashion.
  - DMA layer operations operate on struct device
  - Main include file: <linux/dma-mapping.h>
- Dealing with difficult hardware:
  - If driver knows device is no dma-capable:
    - int **dma_set_mask**(struct device *dev, u64 mask);
      - "mask": Number of bits device can address
      - retval is zero if dma is not possible with mask
  - Not needed for devices supporting 32-bit DMA

- DMA mappings:
  - Combination of:
    - DMA buffer allocation
    - Device-accessible address to buffer
  - Do not use virt_to_bus
  - If available, IOMMU may have "mapping registers"
  - Use of "bounce buffers" used when DMA buffer not device-reachable.
  - DMA layer maintains "cache coherency" with CPU
  - Bus addresses (opaque): dma_addr_t

- DMA mappings for PCI code:
  - Coherent DMA mappings:
    - Mapping must be available to both CPU and peripheral in the same time.
    - Requires setting up mapping for driver's entire lifetime
    - Discouraged use
  - Streaming DMA mappings:
    - Per-transaction mappings:
      - Allocated on transfer init
      - Freed on interrupt receive
    - On some architectures can be faster than coherent mappings
    - Prefered
- Each PCI DMA mapping type dealt with differently

- Setting up coherent DMA mappings:
  - void ***dma_alloc_coherent**(struct device *dev, size_t size, dma_attr_t *dma_handle, int flag);
    - Allocates and maps buffer
    - "dev": device
    - "size": buffer size
    - "dma_handle": bus address if call successfull
    - "flag": GFP_KERNEL or GFP_ATOMIC (if in atomic context).
    - retval is kernel virtual address for use by driver
    - Allocates DMA'able region
    - Smallest allocatable size is one page

- void **dma_free_coherent**(struct device *dev, size_t size, void *vaddr, dma_addr_t dma_handle);
  - All parameters must be properly set
- DMA pools:
  - Allocating small DMA areas
  - <linux/dmapool.h>
  - struct dma_pool ***dma_pool_create**(const char *name, struct device *dev, size_t size, size_t, align, size_t allocation);
    - "name": pool name
    - "dev": device
    - "size": size of buffers in pool

    - "align": hardware alignment for pool buffers (in bytes)
    - "allocation": if nonzero, memory boundary not to be crossed-over by.
    - allocated buffers.
- void **dma_pool_destroy**(struct dma_pool *pool);
- void ***dma_pool_alloc**(struct dma_pool *pool, int mem_flags, dma_addr_t *handle);
    - "pool": pool to allocate from
    - "mem_flags": GFP_*
    - "handle": bus address if call successfull
    - retval is kernel virtual address for use by driver
- void **dma_pool_free**(struct dma_pool *pool, void *vaddr, dma_addr_t addr);

- Setting up streaming DMA mappings:
  - Work on buffer preallocated by driver
  - Must indicate transfer direcation
    - DMA_TO_DEVICE
    - DMA_FROM_DEVICE
    - DMA_BIDIRECTIONAL (may have performance penalty).
    - DMA_NONE (debugging)
  - dma_addr_t **dma_map_single**(struct device *dev, void *buffer, size_t size, enum dma_data_direction direction);
    - retval is bus address to pass to device

- void **dma_unmap_single**(struct device *dev, dma_addr_t dma_addr, size_t size, enum dma_data_direction direction);
  - Parameters must match maping initialization
- Important notes:
  - Can only transfer in specified direction
  - Once mapped, the buffer cannot be touched by the driver, must therefore fill buffer prior to transfer.
  - Buffer should not be unmapped until transfer is complete
- void **dma_sync_single_for_cpu**(struct device *dev, dma_addr_t bus_addr, size_t size, enum dma_data_direction direction);

    - Reacquire buffer for driver manipulation without unmapping it.
  - void **dma_sync_single_for_device**(struct device *dev, dma_addr_t bus_addr, size_t size, enum dma_data_direction direction);
    - Return buffer back to device
- Single-page streaming mappings:
  - May want to transfer buffer for which a struct page is available.
  - Example of user-space buffers mapped using get_user_pages.

- dma_attr_t **dma_map_page**(struct device *dev, struct page *page, unsigned long offset, size_t size, enum dma_data_direction direction);
- void **dma_unmap_page**(struct device *dev, dma_addr_t dma_address, size_t size, enum dma_data_direction direction);
- Avoid partial-page mappings because of potential cache-coherency problems.

- Scatter/gather mappings:
  - Transfering several buffers in the same time
  - Many devices accept list of buffer pointers for single DMA.

- In case of bounce buffer use, separate buffers may be concatenated.
- First set, fill scatter list:
  - Architecture-dependent
  - struct scatterlist: <asm/scatterlist.h>
  - struct page *page;
    - Page containing buffer to be used
  - unsigned int length;
    - Buffer size
  - unsigned int offset;
    - Buffer offset within page
- Second, map scatter buffer:

- int **dma_map_sg**(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);
  - "nents": number of entries in list
  - Function stores attributed DMA bus addresses into scatterlist struct.
- Third, transfer buffers:
  - Loop through scatter list and transfer each buffer
  - Use kernel macros to retrieve dma_addr_t from arch-specific struct scatterlist.
  - dma_addr_t **sg_dma_address**(struct scatterlist *sg);
    - Returns DMA address for given scatterlist entry
  - unsigned int **sg_dma_len**(struct scatterlist *sg);
    - Returns length of buffer
    - May be different from the one set prior to calling dma_map_sg()

- Finally, when done, unmap:
  - void **dma_unmap_sg**(struct device *dev, struct scatterlist *list, int nents, enum dma_data_direction);
    - "nents" is value passed originally to dma_map_sg(), not what that function actually did.
- As with streaming DMA mappings, scatter/gather mappings have strict usage rules.
- To temporarily reaquire buffers, use:
  - void **dma_sync_sg_for_cpu**(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);
  - void **dma_sync_sg_for_device**(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);

- PCI double-address cycle mappings:
  - Kernel supports special mode for 64-bit Double-Address Cycle (DAC) PCI transfers.
  - <linux/pci.h>
  - int **pci_dac_set_dma_mask**(struct pci_dev *pdev, u64 mask);
    - retval 0 if DAC can be used
  - dma64_addr_t **pci_dac_page_to_dma**(struct pci_dev *pdev, struct page *page, unsigned long offset, int direction);
    - DAC mappings should live in high memory
    - DAC mappings must be created one-page at a time

    - "direction": PCI_DMA_TODEVICE, PCI_DMA_FROMDEVICE, PCI_DMA_BIDIRECATIONAL.
- No need for "unmapping" DAC mappings
- Must, however, restrict access:
    - void **pci_dac_dma_sync_single_for_cpu**(struct pci_dev *pdev, dma64_addr_t dma_addr, size_t len, int direction);
    - void **pci_dac_dma_sync_single_for_device**(struct pci_dev *pdev, dma64_addr_t dma_addr, size_t len, int direction);

Hardware access

1. I/O ports and I/O memory
2. I/O registers vs. conventional memory
3. Using I/O ports
4. Using I/O memory
5. Ports as I/O memory

1. I/O ports and I/O memory

- Devices controlled through access to its special registers.
- Registers often placed in consecutive memory addresses.
- Most architectures have single memory address space and device address space.
- Some architectures have separate I/O ports and main memory, either by design of the the architecture or design of the actual board. The x86 is an example architecture where I/O ports exist.

- Kernel provides separate functions for dealing with I/O ports and dealing with I/O memory.

2. I/O registers vs. conventional memory

- Compiler, runtime optimizations by the CPU, and caching are fine for memory accesses, but they aren't appropriate for access to device registers.
- Drivers must make sure no such optimization takes place when reading and writing to device registers.
- Hardware caching problem usually taken care of either in hardware, prior to Linux's boot or by Linux initialization code.

- Must use a "memory barrier" to protect against compiler and CPU optimizations.
- Compiler barrier:
  - <linux/kernel.h>
  - void **barrier**(void);
    - Tell compiler to insert memory barrier (i.e. store all values to memory) and reread if need be.
    - Has no effect on possible hardware optimizations
- Hardware barriers:
  - Platform-dependent
  - <asm/system.h>

- void **rmb**(void);
  - Complete all reads appearing before statement prior to executing any subsequent read.
- void **read_barrier_depends**(void);
  - Less stringent "rmb()". Block reordering of reads depending on data from other reads.
  - Discouraged. Use rmb() instead.
- void **wmb**(void);
  - Complete all writes appearing before statement prior to executing any subsequent write.
- void **mb**(void);
  - Do both rmb() and wmb().

- SMP-equivalent barriers:
  - void **smp_rmb**(void);
  - void **smp_read_barrier_depends**(void);
  - void **smp_wmb**(void);
  - void **smp_mb**(void);
- Barriers should be inserted in between read/write operations.
- Combined setting of value and memory barrier macros:
  - **set_mb**(var, value)

  - **set_wmb**(var,value)
  - **set_rmb**(var,value)
- Barriers are used in kernel's locking mechanisms (spinlocks, atomic_t, etc.)

3. Using I/O ports

- I/O port allocation:
  - Must "allocate" I/O region prior to using it
  - <linux/ioport.h>
  - struct resource ***request_region**(unsigned long first, unsigned long n, const char *name);
    - "first": first port in range
    - "n": number of ports
    - "name": name of device
    - retval non-NULL means allocation success
    - retval NULL means allocation failure
    - Allocated ports visible in /proc/ioports

- void **release_region**(unsigned long start, unsigned long n);
  - Release previously-allocated region
- int **check_region**(unsigned long first, unsigned long n);
  - Check if region is already allocated
  - Deprecated since checking and request are not atomic
  - Use request_region() directly
- Manipulating I/O ports:
  - Some parameter types and return values depend on architecture.

- Input:
  - <arch-dep-type> **inb**(<arch-dep-type> port); /* 8 bit */
  - <arch-dep-type> **inw**(<arch-dep-type> port); /* 16 bit*/
  - <arch-dep-type> **inl**(<arch-dep-type> port); /* 32 bit */
- Output: Performing direct I/O
  - void **outb**(unsigned char byte, <arch-dep-type> port);/* 8 bit */
  - void **outw**(unsigned char byte, <arch-dep-type> port);/* 16 bit */
  - void **outl**(unsigned char byte, <arch-dep-type> port);/* 32 bit */

- I/O port access from user space:
  - On x86/PC, ports are accessible from user-space in certain conditions:
    - Application built with -O
    - Use sys_ioperm or sys_iopl to get permission to use I/O ports.
    - Request for sys_ioperm or sys_iopl must be done while app is running as root or has CAP_SYS_RAWIO.
- String operations:
  - Write entire strings instead of single data units
  - Sometimes processors have special instructions. If not, tight loops are used to write out entire string.

- Careful: single data unit operations do byte-swapping in case peripheral and system have different byte ordering. String operations don't.
- Input:
  - void **insb**(<arch-dep-type> port, void *addr, unsigned long count);
  - void **insw**(<arch-dep-type> port, void *addr, unsigned long count);
  - void **insl**(<arch-dep-type> port, void *addr, unsigned long count);
- Output:
  - void **outsb**(<arch-dep-type> port, void *addr, unsigned long count);

- void **outsw**(<arch-dep-type> port, void *addr, unsigned long count);
- void **outsl**(<arch-dep-type> port, void *addr, unsigned long count);

- Pausing I/O:
  - On some systems, problems if transfers are too fast
  - Must sometimes insert pauses
  - Kernel provides pausing variants of previous calls, they call end with _p (inb_p(), outb_p(), etc.) Such variants are often the same as the original calls.

- Platform dependencies:
  - I/O instructions are highly processor-dependent (if they exist at all).
  - Often the data types are completely different
  - Code conducting I/O is therefore often architecture-dependent as well.
  - Examples:
    - x86 and x86_64:
      - All I/O port functions supported
      - Port numbers are "unsigned short"
    - ARM:
      - Ports are memory-mapped

- All I/O port functions supported
- Port numbers are "unsigned int"

- PPC:
  - Ports are memory-mapped
  - All I/O port functions supported
  - Port numbers are "unsigned char *" for 32-bit and "unsigned long" on 64-bit.
- MIPS:
  - Ports are memory-mapped
  - All I/O port functions supported
  - Port numbers are "unsigned long"

4. Using I/O memory

- I/O memory allocation and mapping:
  - Allocation:
    - <linux/ioport.h>
    - struct resource ***request_mem_region**(unsigned long start, unsigned long len, char *name);
    - retval is non-NULL on success
    - retval is NULL on failure
    - I/O mem allocation visible in /proc/iomem
  - Freeing:
    - void **release_mem_region**(unsigned long start, unsigned long len);

- Checking:
  - int **check_mem_region**(unsigned long start, unsigned long len);
    - Deprecated: can't guarantee atomic check and request
- Mapping:
  - Virtual-memory operation similar to vmalloc() and friends.
  - <linux/vmalloc.h>
  - void ***ioremap**(unsigned long phys_addr, unsigned long size);
    - Map a physical address range to virtual address space
    - Used mostly for mapping PCI devices into kernel's space
    - Returned addresses shouldn't be accessed directly
    - Returned addresses should be accessed using proper I/O functions

- void ***ioremap_nocache***(unsigned long phys_addr, unsigned long size);
  - Similar to ioremap, but makes sure remapped area is not cached
  - In most cases, the same as ioremap
- void **iounmap**(void *addr);
  - Unmap ioremap'ed region

- Accessing I/O memory:
  - Some platforms tolerate direct dereferencing of I/O mem, but this is dangerous, non-portable, and highly discouraged.
  - Helper functions operate on addresses returned by ioremap()

- Read functions:
  - unsigned int **ioread8**(void *addr);
  - unsigned int **ioread16**(void *addr);
  - unsigned int **ioread32**(void *addr);
- Write functions:
  - void **iowrite8**(u8 value, void *addr);
  - void **iowrite16**(u16 value, void *addr);
  - void **iowrite32**(u32 value, void *addr);
- Repeat read functions:
  - void **ioread8_rep**(void *addr, void *buf, unsigned long count);
  - void **ioread16_rep**(void *addr, void *buf, unsigned long count);

- void **ioread32_rep**(void *addr, void *buf, unsigned long count);

- Repeat write functions:
  - void **iowrite8_rep**(void *addr, const void *buf, unsigned long count);
  - void **iowrite16_rep**(void *addr, const void *buf, unsigned long count);
  - void **iowrite32_rep**(void *addr, const void *buf, unsigned long count);
- Memory block operations:
  - void **memset_io**(void *addr, u8 value, unsigned int count);
  - void **memcpy_fromio**(void *dest, void *source, unsigned int count);

- void **memcpy_toio**(void *dest, void *source, unsigned int count);

- Older functions:
  - <arch-dep-type> **readb**(address);
  - <arch-dep-type> **readw**(address);
  - <arch-dep-type> **readl**(address);
  - void **writeb**(<arch-dep-type> value, address);
  - void **writew**(<arch-dep-type> value, address);
  - void **writel**(<arch-dep-type> value, address);
  - Confusion over "b", "w", "l": moving to u8, u16, u32, u64

5. Ports as I/O memory

- In some cases, the same piece of hardware can sometimes appear as accessible via I/O ports and in other instances (other implementations) appear as memory-mapped.
- Kernel provides functions allowing I/O ports to be accessed as I/O memory.
- So:
  - Use request_region() to reserve I/O port region
  - Remap I/O port region to memory:
    - void ***ioport_map**(unsigned longport, unsigned int count);

- Can then use ioread*() and iowrite*() instead of port functions
- Unmapping:
  - void **ioport_unmap**(void *addr);

Char drivers

1. Major and minor
2. The file operations structure
3. The file structure
4. The inode structure
5. Char device registration
6. The older way for device registration
7. Open and release
8. Read and write
9. Read

10. Write
11. readv/writev
12. ioctl
13. Blocking I/O
14. poll and select
15. fsync
16. fasync
17. llseek
18. Access control on a device file

1. Major and minor

- Basics:
  - Char devices accessed through names in fs
  - Device nodes typically in /dev
  - "c"-type files for char
  - "b"-type files for block
  - Use "ls -l" to see device types in /dev
  - Major/minor tuples for each device
  - One major number = one type of device
  - Minor number = device instance
  - Minor number is ignored by kernel

- The internal representation of device numbers:
  - dev_t
  - Macros in <linux/kdev_t.h>:
    - MAJOR(dev_t dev);
    - MINOR(dev_t dev);
    - MKDEV(int major, int minor);
  - Currently, dev_t is 32 bit:
    - 12-bit major number
    - 20-bit minor number

- Allocating and freeing device numbers:
  - **register_chrdev_region**(): static allocation
    - Begining device number for range
    - Number of devices in range (minor numbers)
    - Name of device
  - **alloc_chrdev_region**(): dynamic allocation
    - Pointer to dev_t
    - First minor number, usually 0
    - Number of devices in range
    - Name of devices

- **unregister_chrdev_region**(): deallocation
  - Begining device number for range
  - Number of devices in range
- Dynamic allocation of major numbers:
  - Static numbers found in Documentation/devices.txt
  - No new numbers allocated
  - Strongly encouraged to use alloc_chrdev_region()
  - Problem with node creation beforehand
  - See /proc/devices or, if driver exports info in sysfs, the relevant directory entry in sysfs.
  - Example /proc/devices parsing script p.46 of LDD3

- Could also count on exact same dynamic number being reallocated if same driver loading sequence. This, though, may become deprecated as some kernel developers have hinted.
- Can write code that supports both static and dynamic allocation. See p.48 of LDD3.

2. The file operations structure

- Basics:
  - Connection between major/minor nbrs and char driver callbacks:
    - struct file_operations
  - Defined in <linux/fs.h>
  - Bunch of function callbacks
  - Each opened file (struct file) has an associated sturct file_operations:
    - f_op
  - file_operations are the actual implementation of the main filesystem calls: open, close, read, write, etc.

- One callback per implemented call
- NULL for unsupported calls:
  - Kernel behavior for NULL depends on call
- Use of __user to specify that pointer in declaration is from user_space and shouldn't be used directly.

- Details:
  - struct module *owner;
    - THIS_MODULE
  - loff_t (***llseek**) (struct file *, loff_t, int);
    - Set read/write position
    - retval is new position

    - Position counter unpredictable if NULL
- ssize_t (*__read__) (struct file *, char __user *, size_t, loff_t *);
    - Read from device
    - If NULL, -EINVAL
    - retval is how much data read
- ssize_t (*__aio_read__) (struct kiocb *, char __user *, size_t, loff_t);
    - Asynchronous read
    - May return even incomplete
    - If NULL, normal read gets called

- ssize_t (*__write__) (struct file *, const char __user *, size_t, loff_t *);
  - Write to device
  - If NULL, -EINVAL
  - retval is how much data written
- ssize_t (*__aio_write__) (struct kiocb *, const char __user *, size_t, loff_t);
  - Asynchronous write
- int (*__readdir__) (struct file *, void *, filldir_t);
  - Should be NULL
  - For FS implementations only

- unsigned int (*__poll__) (struct file *, struct poll_table_struct *);
  - One of poll, epoll or select
  - Query if read or write would block
  - If NULL, device is non-blocking on either read or write
- int (*__ioctl__) (struct inode *, struct file *, unsigned int, unsigned long);
  - Device-specific commands
  - Some ioctl commands recognized by kernel directly without calling ioctl() callback.
  - If NULL, -ENOTTY

- int (*__mmap__) (struct file *, struct vm_area_struct *);
  - Mapping of device to process address space
  - If NULL, -ENODEV
- int (*__open__) (struct inode *, struct file *);
  - First op on device (always)
  - Not required
  - If NULL, open always succeeds, but driver is never notified.

- int (*__flush__) (struct file *);
  - Called when file descriptor closed and should execute and wait for any outstanding operation.
  - Per-file operation
  - Not the same as fsync
  - Very seldom used
  - If NULL, app request ignored
- int (*__release__) (struct inode *, struct file *);
  - Called when file struct is released (file is closed)
  - Like open, can be NULL
- int (*__fsync__) (struct file *, struct dentry *, int datasync);

  - Kernel-side of fsync system call
  - Called by user to flush pending data
  - If NULL, -EINVAL
- int (*__aio_fsync__) (struct kiocb *, int datasync);
  - Asynchronous version of fsync
- int (*__fasync__) (int, struct file *, int);
  - To notify device of changes in FASYNC flag
  - See async operations later
- int (*__lock__) (struct file *, int, struct file_lock *);
  - File locking
  - Never implemented by drivers

- ssize_t (*__readv__) (struct file *, const struct iovec *, unsigned long, loff_t *);
  - Scatter/gather read
  - Operations on multiple memory areas
  - If NULL, loop around regular read
- ssize_t (*__writev__) (struct file *, const struct iovec *, unsigned long, loff_t *);
  - Scatter/gather write
  - Operations on multiple memory areas
  - If NULL, loop around regular write

- ssize_t (*__sendfile__) (struct file *, loff_t *, size_t, read_actor_t, void *);
  - Move file content from one file descriptor to another with minimal copy.
  - Drivers usually leave as NULL
- ssize_t (*__sendpage__) (struct file *, struct page *, int, size_t, loff_t *, int);
  - Other half of sendfile
  - Called by kernel to move one page at a time
  - Not usually implemented by drivers

- unsigned long (*__get_unmapped_area__)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
  - Find proper location in process address to map device
  - Usually done by mm code
  - Available for drivers with special alignment requirements
  - Usually left as NULL
- int (*__check_flags__)(int);
  - Check flags passed to an fcntl (manipulate file descriptor)
- int (*__dir_notify__)(struct file *filp, unsigned long arg);
  - Call when uspace uses fcntl to request directory change notifications.

    - Usually implemented by FSes only
- Most important calls:
    - owner
    - open
    - release
    - read
    - write
    - ioctl
    - llseek

- Usefull calls:
  - mmap
  - poll

3. The file structure

- "struct file" defined in <linux/fs.h>
- Unrelated to userspace FILE pointers
- Each is entirely opaque to the oposite space
- Represents an open file
- Created by kernel and passed to open
- Released once all file instances are closed
- Relevant fields (others not relevant to device drivers):

- modet_t f_mode:
  - FMODE_READ: Readable
  - FMODE_WRITE: Writable
  - Permissions checked by kernel prior to file-op invocation
- loff_t f_ops:
  - Current read/write position
  - Should not be modified by read/write file-ops
  - Use last argument of callback instead
  - Can be modified by llseek
- unsigned int f_flags:
  - O_RDONLY, O_NONBLOCK, O_SYNC
  - Driver should usually just check for O_NONBLOCK

- struct file_operations *f_op: The file-ops for the file
  - The file-ops for the file
  - Assigned at open by kernel
  - Could be reassigned at runtime by driver to implement different behavior under the same major number.
- void * private_data:
  - Set to NULL on open
  - Useful to preserve "private" information across syscalls
  - Careful to free on release()
- struct dentry *f_dentry:
  - Directory entry associated with file
  - Not typically useful for drivers

- Can be used to obtain inode struct:
  - filp->f_dentry->d_inode

4. The inode structure

- Kernel internal representation of files
- Can be many file structs pointing to multiple open file descriptors, but all pointing to same inode.
- inode struct contains a lot of info.
- Relevant fields:
  - dev_t i_rdev:
    - For device driver inodes, this is device number
  - struct cdev *i_cdev:
    - Kernel internal representation of char devices

    - When inode is char device, points to actual struct cdev
- Should not access inode fields directly, use macros instead:
  - unsigned int **iminor**(struct inode *inode)
  - unsigned int **imajor**(struct inode *inode)

5. Char device registration

- Must allocate struct cdev so kernel can call char device callbacks.
- Use <linux/cdev.h>
- Allocating single cdev struct:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

- Initializing existing cdev struct:
  - void **cdev_init**(struct cdev *cdev, struct file_operations *fops);

- Typically, should use a device-specific struct to hold info about your device. Something like my_device_struct, which contains a struct cdev as one of its elements.
- Must set owner (for both static and dynamic):

```
my_cdev->owner = THIS_MODULE;
```

- Must add device to system (for both static and dynamic):
  - int **cdev_add**(struct cdev *dev, dev_t num, unsigned int count);
  - the cdev struct

  - the first device number to which device responds
  - how many devices are associated with device
- About cdev_add:
  - Can fail => Must check retval
  - Once called, device is live and callback may be invoked.
  - Should not be called until device is fully initialized
- To remove:
  - void **cdev_del**(struct cdev *dev);
- Example device registration on p.57

6. The older way for device registration

- int **register_chrdev**(unsigned int major, const char *name, struct file_operations *fops);
  - Major number (0 for dynamically allocated)
  - Device name
  - file-ops
- int **unregister_chrdev**(unsigned int major, const char *name);
  - Same major and name passed to register_chrdev.

7. Open and release

- open should do:
  - Identify the device being opened, if using new registration scheme:
    - Assuming the cdev used in device is a member of a device-specific struct such as my_device_struct, and was initialized and registered using something like:

```
cdev_init(&my_device_instance->cdev, ...);
cdev_add(&my_device_instance->cdev, ...);
```

    - We can now obtain the pointer to my_device_instance by substracting from the inode->i_cdev pointer the offset of the cdev element in the my_device_struct struct.

- **container_of**(pointer, container_type, container_filed) macro:
  - pointer to struct member (inode->i_cdev)
  - type of container (struct scull_dev)
  - name of member within container (cdev)
- Set filp->private_data to pointer returned by container_of() for other fileops to use.

- If using old registration scheme:
  - Use iminor() macro to get device minor number
  - Check if device is one of those supported
- Other open responsibilities:
  - Check for device-specific errors
  - Initialize the device if first open

    - If necessary, update f_op
    - Allocate and fill any private data for fip->private_data
  - Example open on p.59
  - If O_NONBLOCK specified, open should return without wait. Some open functions may indeed have lengthy initialization procedures.
- release should do:
  - release is what happens on close()
  - reverse open:
    - Deallocate anything allocated in filp->private_data
    - Shut down device on last close

- Example basic release on p.59

- Generalities about open/release:
  - Not all processes that have a reference to a given struct file have gotten it through open(), yet they all call close() on the struct file on exit. How does this work?
  - Calls to fork or dup do not result in new struct file, they just increment a counter in the existing struct file.
  - Conversely, the counter is decremented on close().
  - release() will be called by the kernel only when the increment reaches zero.

- flush() called every time there is a close(), though few drivers support this call.

8. Read and write

- ssize_t **read**(struct file *filp, char __user *buff, size_t count, loff_t *offp);
- ssize_t **write**(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
  - filp = file pointer
  - count = size of data to transfer
  - buff = buffer to transfer to (read) or from(write)
  - offp = offset at which file is being accessed
- A lit bit more about buff:
  - User-space pointer

- May not even be "visible" from kernel space in some architectures,
- May point to area that is paged out
- May be sent by malicious program

- Must use appropriate functions for user-space references:
  - read/write should use copy_to_user/copy_from_user respectively.
  - read/write operations should update *offp
  - Any update on *offp will be propagated by the kernel to file struct.

- read/write should return negative value on error
- read/write should return 0 or positive value to indicate number of bytes successfully transferred.
- If partial success, read/write should first return the number of bytes transferred, and return error on next call.
- If retval < 0, user-space sees -1, and must use errno to get error.

9. Read

- Interpretation of return value by user-space app:
  - bytes == count, transfer success
  - count > bytes > 0, application retries
  - bytes == 0, end-of-file
  - bytes < 0, error => errno from <linux/errno.h>
- In case of data not present, read() should typically block.
- Example implementation p.67.

10. Write

- Interpretation of return value by user-space app:
  - bytes == count, transfer success
  - count > bytes > 0, application retries
  - bytes == 0, no error, therefore retries
  - bytes < 0, error => errno from <linux/errno.h>
- Example implementation p.68-69.

11. readv/writev

- Vector operations
- Vector entries contain: buffer pointer + length value.
- If missing, read/write are called multiple times by kernel.
- If useful, always best to have real readv/writev
- Easiest, have a loop in driver calling read/write
- If implemented, should be more fancy, like having command reodering.
- Not typically implemented by driver

12. ioctl

- Background:
  - Must be able to control device with more than just read/write.
  - User-space prototype:
    - int **ioctl**(int fd, unsigned long cmd, ...);
  - Not actually a variable number of arguments, instead, 3rd arg is:
    - char *argp
  - The "..." avoids typechecking at build time
  - argp can be used to pass all sorts of things

- ioctl() often considered a nuisance by kernel developers because it opens undocumented/unauditable back-doors to drivers.
- Driver prototype:
  - int (*__ioctl__) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
  - inode and filp are same as passed to open()
  - cmd passed as-is from user-space
  - arg is passed as "unsigned long" regardless of its type
- Usually implemented as a huge "switch(cmd)"
- Each "cmd" is interpreted and acted on differently

- Commands must be defined in headers, and shared with user-space apps so that they know what commands to invoke for a given action.

- Choosing the ioctl commands:
  - Numbers should be unique to avoid mistaken driver access to cause damage.
  - Use of "magic numbers"
  - To help managing these numbers, the command codes have been split up in several bitfields.
  - include/asm/ioctl.h defines the separate bitfields
  - Documentation/ioctl-number.txt defines the already allocated "magic" numbers.

- Bitfields:
  - Type (magic number), unique to driver: 8 bits
  - Ordinal number, unique command "id" in driver: 8 bits
  - Direction of transfer: bit-mask
    - _IOC_NONE: no transfer
    - _IOC_READ: read from device
    - _IOC_WRITE: write to device
    - _IOC_READ | _IOC_WRITE: write to and read from device
  - Size of argument: usually 13 or 14 bits, but actual width depends on arch.
    - Not mandatory, but recommended
    - If need larger data structs, can ignore this field
- Macros for defining command numbers:
  - <asm/ioctl.h>

- _IO(type,nr)
- _IOR(type,nr,datatype)
- _IOW(type,nr,datatype)
- _IOWR(type,nr,datatype)
- type and nr (number) explicit, size obtained using sizeof(datatype)

- Example:

```
#define MY_IOC_MAGIC          251
#define MY_IOCFIRSTCOMMAND    _IO(MY_IOC_MAGIC, 0)
#define MY_IOCSECONDCOMMAND   _IO(MY_IOC_MAGIC, 1)
#define MY_IOCREAD            _IOR(MY_IOC_MAGIC, 2, int)
#define MY_IOCWRITE           _IOW(MY_IOC_MAGIC, 3, int)
#define MY_IOCREADWRITE       _IOWR(MY_IOC_MAGIC, 4, int)
#define MY_IOC_MAXNR          10
```

- Actual command definitions not interpreted by kernel.
- Could, therefore, define entirely different command number ranges.

- The return value
  - Usually depends on actual outcome of switch() statement.
  - By default, most kernel functions return -EINVAL
  - According to POSIX, should return -ENOTTY, which would be interpreted by the C library as meaning "inappropriate ioctl for device".

- The predefined commands
  - Though ioctl commands are typically not interpreted by the kernel, some are.
  - Interpreted commands are recognized prior to the invocation of your driver's ioctl().
  - If driver defines same numbers, driver will not see commands.
  - 3 types of predefined commands according to what they can be issued on:
    - Any file (the only one relevant to device drivers, magic nbr "T").
    - Regular files

- Filesystem-specific files
- Commands specified for any file, including drivers:
  - FIOCLEX:
    - File descriptor closed on execution of new program
  - FIONCLEX:
    - Undo FIOCLEX
  - FIOASYNC:
    - Set or reset async notification on file (unused)
  - FIOQSIZE:
    - Get size of file or directory. ENOTTY on devices
  - FIONBIO:
    - Modify the O_NONBLOCK flag in filp->f_flags. Last ioctl() argument serves to indicate whether to enable or disable blocking.

- Using the ioctl argument:
  - If not pointer, use as-is
  - If pointer, must make sure address is valid:
    - Can use copy_to_user() and copy_from_user()
    - Since ioctl() transfers, there are other functions that can used as well.
    - Verify if region is valid user space:
      - int **access_ok**(int type, const void *addr, unsigned long size);
      - type is either VERIFY_READ, from uspace, or VERIFY_WRITE, to uspace.
      - addr is user-space address
      - size is ... well, hmmm, size
      - retval is 1 on sucess, 0 on failure
      - Should return -EFAULT on failure

- access_ok() checks to make sure address is not kernel-space
- Transfer macros:
  - <asm/uaccess.h>
  - **put_user**(datum, ptr) and **__put_user**(datum, ptr)
    - Writes datum to use-space
    - Size of transfer determined using ptr type
    - __put_user assumes having already called access_ok()
    - retval is zero on success
  - **get_user**(local, ptr) and **__get_user**(local, ptr)
    - Read datum from user-space into "local"
    - retval is zero on success

- Capabilities and restricted operations
  - Typically access is controled using file-permissions
  - Regardless of actual file-permissions, some operations should not be allowed to all users.

- In addition to simple user permissions, Linux provides "capabilities".
- Capabilities enable selectively setting priviliges with finer granularity than just using root/non-root.
- Capabilities == permission management
- Capabilities controled using sys_capget() and sys_capset().
- Available capabilities defined in <linux/capability.h>
- No additional capabilities can be defined
- Capabilities relevant to drivers:

- CAP_DAC_OVERRIDE:
  - Override access restrictions on files and directories
- CAP_NET_ADMIN:
  - Administer networking
- CAP_SYS_MODULE:
  - Load/unload modules
- CAP_SYS_RAWIO:
  - Perform raw I/O
- CAP_SYS_ADMIN:
  - "Administer" system (many things)
- CAP_SYS_TTY_CONFIG:
  - TTY configuration

- Drivers should check for capabilities prior to carrying out privileged operations.

- Capability checking:
  - <linux/sched.h>
  - int **capable**(in capability);
  - Returns 1 if capable, 0 otherwise
  - Driver could return -EPERM on failure
- Device control without ioctl:
  - Can use "control sequences" written to device, like escape sequences on console.
  - Control sequences require parsing input to device, and driver must make sure that such sequences are not written to actual device.

- Should be used if device doesn't actually transfer any data, but just acts on commands.
- If ASCII commands, can even use "cat" to send commands to device, which avoids having to implement an ioctl() and a custom app to talk to driver.

13. Blocking I/O

- What's blocking I/O:
  - What if driver can't satisfy a read() or write()?
  - Driver must be able to put calling process to sleep
- Introduction to sleeping:
  - Putting process in special mode and taking it off the scheduler's queue.
  - Rules:
    - Never sleep in a critical section
    - Never sleep with interrupts disabled
    - On wakeup, no time or event context => recheck sleep condition.
    - Never sleep if waking procedure unknown/unsure

- A "wait queue" contains list of sleeping processes
- Process usually woken up as a result of a hardware interrupt.
- copy_to/from_user can sleep => this type of sleeping is ok.

- Introduction to wait queues:
  - Managed using "wait queue head"
  - wait_queue_head_t: <linux/wait.h>
  - Static initialization:
    - DECLARE_WAIT_QUEUE_HEAD(name);

- Dynamic initialization:
  - void **init_waitqueue_head**(wait_queue_head_t *queue);
- Simple sleeping
  - Basic sleeping macros take 2 or 3 arguments:
    - queue, the wait_queue_head_t, not a pointer
    - condition, a boolean expression evaluated by macro before and after sleeping. Could be evaluated an arbitrary number of times.
    - timeout, in case a timeout in jiffies can be specified
  - Sleep macros:
    - **wait_event**(queue, condition)
      - Put process in uninterruptible mode (not recommended)

- **wait_event_interruptible**(queue, condition)
  - Put process in interruptible state (may receive signals)
  - If retval != 0, sleep was interrupted, return -ERESTARTSYS
- **wait_event_timeout**(queue, condition, timeout)
  - Wait for a limited amount of time
  - Always returns 0
- **wait_event_interruptible_timeout**(queue, condition, timeout)
  - Same as wait_event_timeout()

- Wakeup functions, the basics:
  - void **wake_up**(wait_queue_head_t *queue);
    - Wake up all processes sleeping in queue.
  - void **wake_up_interruptible**(wait_queue_head_t *queue);
    - Wake up only processes in state interruptible.

- Blocking and nonblocking operations
  - When should a process be put to sleep?
  - Must match Unix semantics
  - Must check for O_NONBLOCK in filp->f_flags
  - Proper semantics:
    - Assuming there are driver-implemented buffers for read and write, which is the case for most drivers. Input buffer avoids loosing data when no one is reading. Output buffer avoids leaving data in user-space while device is not ready.
    - On read, block if not enough data, wake up when data is available even if incomplete.

- On write, block if not enough space in buffer, wake up when space is available in buffer (possibly due to a previous write completing) event if space is insufficient.
- Wait queue for both blocking operations should be different, one for read and one for write.
- If O_NONBLOCK, return -EAGAIN on failure.

- Advanced sleeping:
  - How a process sleeps:
    - 3 steps:
      - Allocate, initialize and pend wait queue entry
      - Set task state
      - Call scheduler
    - wait_queue_head_t is defined in <linux/wait.h>

- Contains linked list and spinlock

  Wait queue entry is queue entry of type wait_queue_t

```
struct __wait_queue {
        unsigned int flags;
#define WQ_FLAG_EXCLUSIVE       0x01
        struct task_struct * task;
        wait_queue_func_t func;
        struct list_head task_list;
};
```

- This structure is used to know which processes to wake up and how.
- In addition to allocating and initializing this struct properly, must also set the process' state.
- Relevant process states
  - TASK_RUNNING:
    - Task is able to run, but not necessarily running

- TASK_INTERRUPTIBLE:
  - Task is sleeping, but may receive signals
- TASK_UNINTERRUPTIBLE:
  - Task is sleeping and can't receive signals

- To modify process state:
  - void **set_current_state**(int new_state);
- Don't manipulate process struct directly
- Prior to calling scheduler, must check if condition waited on became true. Otherwise, there could be a race condition if you are woken up exactly as you are setting the process to go to sleep because what you were waiting for occured:

```
if (!condition)
        schedule();
```

- If schedule() called, process will return in TASK_RUNNING.

- If not, must reset process state to TASK_RUNNING manually.
- Either way, must remove task from wait queue to avoid being woken up again.
- If sleep condition satisfied between if() and schedule(), then this is ok, process is returned to "TASK_RUNNING" and schedule() will return, though not necessarily right away.
- Once done, must check if code needs to sleep again because the sleep condition isn't fulfilled.

- Manual sleeps
  - Can do previously-described procedure manually, or better use helper functions.

- Static wait queue entry declaration:
  - DEFINE_WAIT(my_wait);
- Dynamic wait queue initialization:
  - void **init_wait**(wait_queue_t *wait);
- Add process to wait queue and set state:
  - void **prepare_to_wait**(wait_queue_head_t *queue, wait_queue_t *wait, int state);
- Can now schedule() after checking if sleep condition was fulfilled.
- Cleaning up taken care of by:
  - void **finish_wait**(wait_queue_head_t *queue, wait_queue_t *wait);
- Make sure don't need to sleep again because sleep condition isn't fulfilled.

- Check if wake up due to signal:
  - int **signal_pending**(struct task_struct *p)
  - Send -ERESTARTSYS in such a case.
- Exclusive waits
  - A wake up event usually wakes up all processes waiting in wait queue.
  - All processes must then recheck wait condition and sleep some more if need be.
  - This is fine if there aren't many processes waiting.
  - Otherwise: "thundering herd" (Apache)
  - Fix: WQ_FLAG_EXCLUSIVE:
    - Processes marked with this flag placed at the end of wait queue
    - Processes without flag place at the beginning of wait queue
    - Kernel stops after waking up the first process marked with flag

    - IOW, all processes not marked (if any) wake up, and only marked wakes up.
  - When to use:
    - Serious resources contention
    - Waking up single process results in proper consumption
  - Must use manual wait helper functions for exclusive waits:
    - void **prepare_to_wait_exclusive**(wait_queue_head_t *queue, wait_queue_t *wait, int state);
- The details of waking up:
  - Typical wakeup behavior controlled by actually wakeup function.
  - Default function is default_wake_function()
  - All drivers should use default

- Full wakeup functions:
  - void **wake_up**(wait_queue_head_t *queue);
    - Already covered
  - void **wake_up_interruptible**(wait_queue_head_t *queue);
    - Already covered
  - void **wake_up_nr**(wait_queue_head_t *queue, int nr);
    - Wake up nr exclusive waiters. If 0, wake all up.
  - void **wake_up_interruptible_nr**(wait_queue_head_t *queue, int nr);
    - Same
  - void **wake_up_all**(wait_queue_head_t *queue);
    - Wakes up all processes, whether exclusive or not.
  - void **wake_up_interruptible_all**(wait_queue_head_t *queue);
    - Same

- void **wake_up_interruptible_sync**(wait_queue_head_t *queue);'
  - Insure that process woken up doesn't get to run prior to this function having returned.
- Most drivers should just use wake_up_interruptible()

14. poll and select

- Introduction to poll and select:
  - Non-blocking I/O applications use poll, select, and epoll to determine if data is ready for consumption.
  - These calls can be used to check entire sets of file descriptors for whether one of them is ready for read/write or wait for one of them to be ready for read/write.
  - poll and select essentially equivalent, implemented by 2 separate unix teams in parallel.
  - epoll is Linux-specific for handling thousands of file descriptors.

- In-driver callback for poll, select and epoll:
  - unsigned int (***poll**) (struct file *filp, poll_table *wait);
- Driver should do:
  - Tell kernel which wait queues the process could wait on if it tried to read or write:
    - void **poll_wait**(struct file *, wait_queue_head_t *, poll_table *);
  - Return bitmask of operations that could be performed without blocking.
- What the kernel does:
  - Allocate poll_table to record all wait queues process could wait on.

- For every file descriptor set in user-space for poll(), call its kernel-side callback, which could be your driver, another driver or a filesystem, and provide it with the allocated poll_table.
- Depending on the bitmask returned by the various callbacks invoked by the kernel:
  - If none indicates that I/O can occur without blocking, the user-space poll will sleep until one of the wait queues it places the calling process on wakes that process up.
  - If one of them has something, report it to user-space immediately.
- On wakeup, remove process **from all wait queues** and free allocated poll_table.
- Pass poll_table as NULL to callbacks:
  - If user-space set poll timeout to 0
  - If one of the callbacks for any of the FDs indicated nonblocking I/O is possible.

- For epoll:
  - Avoid allocation/deallocation on every call, since this is expensive when large amounts of FDs.
  - Preallocate poll_table for all activities
  - Release poll_table once done for real
- Flags recognized as part of bitmask:
  - <linux/poll.h>
  - POLLIN:
    - Can read without blocking
  - POILRDNORM:
    - "normal" data available for read. Usually: POLLIN | POLLRDNORM.
  - POLLRDBAND:
    - "out-of-band" data available. Typically unused. Sockets only.

- POLLPRI:
  - "high-priority" data (out-of-band) available. Reported as exception by select.
- POLLHUP:
  - End-of-file. Select told device is readable.
- POLLERR:
  - Error on device. On poll, device is readable and writable
- POLLOUT:
  - Can write without blocking
- POLLWRNORM:
  - Same as POLLOUT. Usually: POLLOUT | POLLWRNORM
- POLLWBAND:
  - "out-of-band" can be written to device. Sockets only.

- Interaction with read and write:
  - reading data from the device
    - If data available and read(), return what is available
    - If data available and poll(), return POLLIN | POLLRDNORM.
    - If no data and read(), wait until any data is there
    - If no data, read() and O_NONBLOCK, return -EAGAIN
    - If no data and poll(), return empty mask for read
    - If end-of-file and read(), return 0 immediately
    - If end-of-file and poll(), return POLLHUP
  - writing data to the device
    - If space available and write(), write whatever can be

- If space available and poll(), return POLLOUT | POLLWRNORM.
- If no space and write(), wait until space is available
- If no space, write() and O_NONBLOCK, return -EAGAIN.
- If no space and poll(), return empty mask for write
- If device full and write(), return -ENOSPC

15. fsync

- Flushing pending output
- write() calls should never wait until data is transmitted, with or without O_NONBLOCK.
- Program requiring wait should use fsync().
- If any program requires fsync(), driver should provide it.
- Prototype:
  - int (*__fsync__) (struct file *file, struct dentry *dentry, int datasync);
  - datasync for filesystems only.

- fsync() doesn't return until all data is flushed to device.
- Usually not defined for drivers, except for block drivers.

16. fasync

- Asynchronous notification
- Needed for apps that don't want to routinely poll() or select().
- Instead, application notifies kernel that it wants to receive a SIGIO when something becomes ready for it.
- If application has set async notification for more than one fd, it must then use poll() or select() to determine which one became available.

- Setting up async notification in user-space:

```
signal(SIGIO, my_sig_handler);           /* Set sig-handler */
fcntl(my_fd, F_SETOWN, getpid());        /* Set fd "owner" */
c_flags = fcntl(my_fd, F_GETFL);         /* Get current flags */
fcntl(my_fd, F_SETFL, c_flags | FASYNC); /* Set async notification */
```

- Driver's involvement:
  - fasync() callback invoked on fcntl(... FASYNC)
  - Should send SIGIO when data is available
- fasync() callback responsibility:
  - int **fasync_helper**(int fd, struct file *filp, int mode, struct fasync_struct **fa);

- First 3 arguments taken from parameters of fasync() callback.
- Last argument is driver's own pointer to a struct fasync_struct *.

- Upon data availability, and if async readers:
  - void **kill_fasync**(struct fasync_struct **fa, int sig, int band);
  - First param same as for fasync_helper
  - sig is typically SIGIO
  - band is POLLIN|POLLRDNORM if read available and POLLOUT|POLLWRNORM if write.

- On close should call internal fasync() callback to remove file from list of async listeners:
  - my_fasync(-1, filp, 0);

17. llseek

- Seeking a device
- llseek() callback serves uspace lseek() and llseek().
- If missing, kernel performs operations on filp->f_pos.
- As stated before, should provide llseek
- Must cooperate with read/write
- Must maintain internal counters for seeking
- If device nonseekable, inform kernel on open():

- int **nonseekable_open**(struct inode *inode, struct file *filp);
- Set llseek() callback in fileops to no_llseek (<linux/fs.h>)

18. Access control on a device file

- Single-open devices:
  - Only one process can open device
  - Avoid when possible
  - Can use atomic_t val:
    - Initialize with value of 1
    - Use atomic_dec_and_test() on open and fail if value is not 0 (retval is 1). If fail, use atomic_inc() to erase effect.
    - Use atomic_inc() on close.
- Restricting access to a single user at a time
  - Allows user to open device multiple times

- Must maintain:
  - PID
  - Use count
- Use critical section (spinlock) in open to:
  - On first open (use count zero), record current->uid
  - On other opens, check current->uid, current->euid, and capable(CAP_DAC_OVERRIDE) to see if access should be granted.
  - Increment use count on all successfull opens
- Use critical section (spinlock) in close to:
  - Decrement use count

- Blocking I/O as an alternative to EBUSY
  - Usually unavailable device should return -EBUSY
  - Sometimes, it's better to just wait until the open can succeed, depending on what's the expeced user experience.
  - On open, put process on wait queue until device is available.
  - On release, wakeup waiting processes
- Cloning the device on open
  - For each separate process doing an open creates a new device which that process now deals with independently of other processes.

- Possible only for non-hardware devices
- tty uses something similar to obtain virtual ttys
- Can set policy for new device on "new user", "new terminal", etc.

Block drivers

1. Registration
2. The block device operations
3. Request processing
4. Some other details

1. Registration

- Block driver registration:
  - <linux/fs.h>
  - int **register_blkdev**(unsigned int major, const char *name);
    - "major" = 0 means dynamic allocation
    - Call to this function is optional
    - Does the following:
      - Allocate major number if need be
      - Create entry in /proc/devices
  - int **unregister_blkdev**(unsigned int major, const char *name);

- Disk registration:
  - Block device operations:
    - Equivalent to char dev's struct file_operations
    - struct block_device_operations: <linux/fs.h>
    - int (***open**)(struct inode *inode, struct file *filp);
      - Called on device open
    - int (***release**)(struct inode *inode, struct file *filp);
      - Called on device close
    - int (***ioctl**)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
      - ioctl
      - Block layer recognizes a fair number of ioctl messages prior to them reaching the device.
      - Most likely driver doesn't need to implement this

- int (*__media_changed__)(struct gendisk *gd);
  - Ask driver if removable storage media has changed
- int (*__revalidate_disk__)(struct gendisk *gd);
  - If media changed, this is invoked to set up new media
  - After this is called, kernel re-reads partition table
- struct module *owner;
  - THIS_MODULE
- No read/write operations, use of block I/O requests

- The gendisk structure:
  - Individual disk device representation in kernel
  - struct gendisk: <linux/genhd.h>
  - int major; int first_minor; int minors;
    - Major number, first minor number and number of "disks"

- char disk_name[32];
  - Disk name as seen in sysfs and /proc/partitions
- struct block_device_operations *fops;
  - Struct described above
- struct request_queue *queue;
  - Request queue (will see shortly)
- int flags;
  - Not often used. See LDD3
- sector_t capacity;
  - Not modified by driver directly
  - Number of 512-byte sectors on disk
- void *private_data;
  - Driver-internal data
- struct gendisk contains a kobject

- Allocating disks:
  - struct gendisk ***alloc_disk**(int minors);
- Freeing disks:
  - void **del_gendisk**(struct gendisk *gd);
- Adding allocated disk to system:
  1. Properly initialized allocated struct gendisk
  2. void **add_disk**(struct gendisk *gd);
     - Should not be called until driver is fully functional as it will result in calls to your driver's callbacks.
     - IOW, a block I/O request callback must have been registered prior to add_disk() being invoked.

- A note on sector sizes:
  - Kernel sees device as flat array of sectors
  - Kernel considers every sector to have 512 bytes

- Can nevertheless override default by modifying parameter in request queue:
  - blk_queue_hardsect_size(dev->queue, hardsect_size);
- Must continue translating sector number nonetheless:
  - set_capacity(dev->gd, nsectors*(harsect_size/512));

2. The block device operations

- The open and release methods:
  - int (*__open__)(struct inode *inode, struct file *filp);
    - inode->ib_dev->bd_disk contains appropriate struct gendisk pointer.
    - Tasks:
      - Set up DMA channels
      - Start disk spinning etc.
    - When is this called:
      - uspace filesystem format
      - uspace partitioning
      - uspace filesystem check
      - kspace filesystem mounting
    - No way for driver to know difference between uspace and kspace open.

- int (*__release__)(struct inode *inode, struct file *filp);
  - Reverse open()
- The ioctl method:
  - May use this to provide uspace with disk geometry info.
  - Not useful to kernel (since it considers disk to be array of sectors).
  - Useful to fdisk

3. Request processing

- Introduction to the request method:
  - Core of block I/O
  - Kernel provides many levels of abstraction to allow both simple drivers and drivers geared towards performance.
  - void **request**(request_queue_t *queue);
    - Called by kernel for every request
    - Typically starts the request processing
  - Every block driver must have request queue
  - Request function "consumes" requests from driver's queue.

- Request function called in atomic context (use of lock provided by driver).
- No requests queued while request function running
- Possible optimization: drop lock in request fct and reacquire prior to return.
- Call to request fct asynchronous to uspace request
- Requests queued up by kernel provide all the context required to be properly processed.

- A simple request method:
  - Requests are stored in struct request

- Request queue traversal is done using **elv_next_request**():
  - Returns NULL when no more requests
- Requests must be explicitely removed from queue using:
  - **blkdev_dequeue_request**(struct request *req);
- Once request is fully processed, must tell kernel that it is so:
  - **void end_request**(struct request *req, int succeeded);
- Some requests are not for actual transfers, but for device command.

- Use **block_fs_request**() to check whether a request is an actual transfer.
- Fields in request:
  - sector_t sector;
    - Beginning sector for request
  - unsigned long nr_sectors;
    - Number of sectors to transfer
  - char *buffer;
    - Kernel virtual address of buffer to be transferred
- To figure out transfer direction, use macro:
  - **rq_data_dir**(struct request *req);
- Request queues are ordered for faster access
- Driver request handler should take advantage of this. More details below.

- Request queues:
  - Basics:
    - struct request_queue or request_queue_t: <linux/blkdev.h>
    - Stores requests
    - Is used by kernel during the creation of requests. Stores:
      - Maximum allowable size
      - Nbr of independent segments part of request
      - Hardware sector size
      - Alignment requirements
      - etc.
    - Properly configured queues should present only valid requests.
    - Request queues allow pluggable I/O schedulers

- Typical I/0 scheduler:
  - Store requests
  - Sort requests
  - Reorder for best performance
  - Merge adjacent requests
- Available schedulers:
  - Deadline scheduler: service request in less than X
  - Anticipatory scheduler: stall device in expectation of repeat read
- Queue creation and deletion:
  - request_queue_t ***blk_init_queue**(request_fn_proc *request, spinlock_t *lock);
    - "request": Request callback
    - "lock": Lock to be held while servicing queue
    - Should be called during driver initialization
  - Set ->queuedata to private data

- void **blk_cleanup_queue**(request_queue_t *req_queue);
  - No more requests queued after this call
  - Should be called during driver finalization
- Queuing functions:
  - struct request ***elv_next_request**(request_queue_t *queue);
    - Return next available request in queue
    - Returns NULL if no more requests
    - Request is marked as "active"
    - Request **not** dequeued
  - void **blk_dequeue_request**(struct request *req);
    - Remove request from queue
  - void **elv_requeue_request**(request_queue_t *queue, struct request *req);
    - Requeue request

- Queue control functions:
  - Control how queue operates
  - void **blk_stop_queue**(request_queue_t *queue);
    - Suspend queue
  - void **blk_start_queue**(request_queue_t *queue);
    - Resume queue
  - void **blk_queue_bounce_limit**(request_queue_t *queue, u64 dma_addr);
    - Set highest physical address for DMA
    - If request comes in from higher address, kernel will use bounce buffer.
  - void **blk_queue_max_sectors**(request_queue_t *queue, unsigned short max);
    - Maximum number of sectors per request

- void **blk_queue_max_phys_segments**(request_queue_t *queue, unsigned short max);
  - Maximum number of non-contiguous memory ranges handled by driver per request.
- void **blk_queue_max_hw_segments**(request_queue_t *queue, unsigned short max);
  - Maximum number of non-contiguous memory ranges handled by device per request.
- void **blk_queue_max_segment_size**(request_queue_t *queue, unsigned int max);
  - Maximum size for segments in a request
- **blk_queue_segment_boundary**(request_queue_t *queue, unsigned long mask);
  - Set maximum memory boundary for requests serviceable by device.

- void **blk_queue_dma_alignment**(request_queue_t *queue, int mask);
  - DMA alignment constraints
  - Requests will match size and alignment
- void **blk_queue_hardsect_size**(request_queue_t *queue, unsigned short max);
  - Specify sector size other than default 512 bytes
  - Kernel will continue to operate on 512 byte assumption however

- The anatomy of a request:
  - Basics:
    - Requests are made up of a set of segments scattered in memory.
    - Kernel may combine adjacent requests, but never joins reads and writes.

- Requests are a linked list of block I/O operations (bio struct)
- Block I/O operations are array of segments to be transfered

- The bio structure:
  - Block I/O requests, regardless of their origin, are packaged in bio struct.
  - bio is then fed to block I/O subsystem which may combine it to other bios to form a request.
  - struct bio: <linux/bio.h>
  - sector_t bi_sector;
    - First sector to be transfered
  - unsigned int bi_size;
    - Bytes to be transfered

- unsigned long bi_flags;
  - Bio operation description
- unsigned short bio_phys_segments;
  - Number of physical segments in bio
- unsigned short bio_hw_segments;
  - Number of segments seen by hardware after DMA mapping
- struct bio *bi_next;
  - Next bio in list
- struct bio_vec bi_io_vec;
  - Array
  - Actual description of the region to be transferred
  - Per-page description
  - This is the data that the driver needs to loop through and write out.

- struct bio_vec

```
struct bio_vec {
        struct page *bv_page;
        unsigned int bv_len;
        unsigned int bv_offset;
};
```

- See fig 16-1 in LDD3, p.482
- Looping around entries in bio entries:
  - **bio_for_each_segment**(bvec, bio, segno) {}
    - bvec: current bio_vec entry
    - segno: segment number
- Can use bio_vec struct entries to create DMA mappings
- If direct page access needed:
  - char *__**bio_kmap_atomic**(struct bio *bio, int i, enum km_type type);
  - void __**bio_kunmap_atomic**(char *buffer, enum km_type type);

- Helper functions:
  - Operate on buffer to be transfered next within bio
  - May not be useful if driver wants to want bio list before deciding what to transfer.
  - struct page ***bio_page**(struct bio *bio);
    - Get pointer to next page to transfer
  - int **bio_offset**(struct bio *bio);
    - Get page offset of request
  - int **bio_cur_sectors**(struct bio *bio);
    - Get number of sectors to transfer from page
  - char ***bio_data**(struct bio *bio);
    - Get kernel logical address to buffer to be transfered
    - Not high mem (at least, by default, block I/O layer doesn't hand highmem buffers to drivers).
  - char ***bio_kmap_irq**(struct bio *bio, unsigned long *flags);
    - IRQ-safe mapping of any type of buffer, even from high-mem.

- void **bio_unmap_irq**(char *buffer, unsigned long *flags);
  - Undo mapping done with bio_kmap_irq

- Request structure fields:
  - struct request internals
  - sector_t hard_sector;
    - First sector not yet transferred
  - unsigned long hard_nr_sectors;
    - Number of sectors yet to transfer
  - unsigned int hard_cur_sectors;
    - Number of sectors left in current bio
  - struct bio *bio;
    - List of bios
  - char *buffer;
    - Kernel logical address of the current buffer to be transferred

- unsigned short nr_phys_segments;
  - Number of non-contiguous physical segments in this request
- struct list_head queuelist;
  - Link to the rest of the request queue
  - Cannot be used as-is until request has been removed from queue

- Barrier requests:
  - May need to force non-reordering of certain operations because some apps depend on it.
  - Examples: databases, journaling filesystems
  - Use of barrier request: request with REQ_HARDBARRIER flag.
  - Must make sure drive's caching doesn't hide non-committing.

- Informing block layer that driver handles barrier requests:
  - void **blk_queue_ordered**(request_queue_t *queue, int flag);
    - Nonzero flag means "can handle"
  - Figuring out whether request contains barrier:
    - int **blk_barrier_rq**(struct request *rq);
      - retval nonzero means this is a barrier request
- Nonretryable requests:
  - Block drivers typically retry failed requests
  - In some cases, the kernel's block layer doesn't want that
  - First, test if kernel wants retry:
    - int **blk_noretry_request**(struct request *req);
      - If retval nonzero, driver should abort

- Request completion functions:
  - Basics:
    - int **end_that_request_first**(struct request *req, int success, int count);
      - Tell block layer, "count" bytes have been transfered since last call.
      - Completion must be signaled in order, even if actual transfers happened out of order.
      - retval is 0 if all sectors have been transferred and request is done
      - If request done, use blkdev_dequeue_request to dequeue request
    - void **end_that_request_last**(struct request *req);
      - Inform anything waiting on request that request is done

- Typically:

```
if (!end_that_request_first(req, 1, sectors_xferred) {
    blkdev_dequeue_request(req);
    end_that_request_last(req);
}
```

- Working with bios:
  - Request callback should:
    - Use rq_for_each_bio(bio, req) {} to go through bios
    - For each bio, use bio_for_each_segment(bvec, bio, segno) {} to go through all segments in bio.
    - Map each bio segment
    - Transfer bio segment
- Block requests and DMA:
  - Instead of going through bios and setting up DMA for each transfer:

    - int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct scatterlist *list);
    - "list": Preallocated scatter list that has as many entries as there are physical segments in the request.
    - Scatterlist returned can have dma_map_sg() used on its entries
  - If no request concatenation wanted:

```
clear_bit(QUEUE_FLAG_CLUSTER, &queue->queue_flags);
```

- Doing without a request queue:
  - Need for driver to do its own sorting/reordering, if any
  - Solid-state storage devices do no benefit from access reordering.
  - Same with RAID
  - Instead of using request callback, use make_request:
    - typedef int (**make_request_fn**)(request_queue_t *q, struct bio *bio);

- make_request can:
  - Execute transfer right away (loop through "bio" list provided)
  - Pend request to other device
- Signaling bio completion directly:
  - void **bio_endio**(struct bio *bio, unsigned int bytes, int error);
  - Do not use on regular request-type bios
- make_request should return 0, regardless of the transfer status.
- If make_request retval nonzero, bio is resubmitted
- To use make_request, must manually allocate queue:
  - request_queue_t ***blk_alloc_queue**(int flags);
    - Allocated queue not set up to receive actual requests
    - "flags": usually GFP_KERNEL
- Must also set up make_request callback:
  - void **blk_queue_make_request**(request_queue_t *queue, make_request_fn *func);

4. Some other details

- Command pre-preparation:
  - Get information about pending request prior to it being returned by elv_next_request().
  - void **blk_queue_prep_rq**(request_queue_t *queue, prep_rq_fn *func);
  - By default no pre-preparation involved
  - typedef int (**prep_rq_fn**)(request_queue_t *queue, struct request *req);
  - See LDD3 for full details

- Tagged command queueing:
  - Some hardware supports having multiple requests active in parallel.
  - If so, tags can be associated with requests so that hardware may tell which one has completed.
  - See LDD3 for full details on how to set up and use Tagged Command Queuing (TCQ).

Network drivers

1. Basics
2. Connecting to the kernel
3. The net_device structure in detail
4. Opening and closing
5. Packet transmission
6. Packet reception
7. The interrupt handler
8. Receive interrupt mitigation
9. Changes in link state

10. The socket buffers
11. MAC address resolution
12. Custom ioctl commands
13. Statistical information
14. Multicast
15. A few other details

1. Basics

- Network devices have no /dev entries
- Use of sockets to talk over network
- Receives data from outside world
- Networking layer supports lots of configurability and stats accumulation.
- Linux net subsystem is network and hardware protocol independent.
- Network stack makes it very easy to use IP and Ethernet protocols.

2. Connecting to the kernel

- Device registration:
  - Like other types of devices, network drivers must initialize things at startup.
  - Unlike char and block devices, there are no major or minor numbers to register.
  - Instead, a network driver must register every interface it detects as part of the system-wide list of recognized interfaces.
  - struct net_device: <linux/netdevice.h>
  - struct net_device is a kobject

- struct net_device ***alloc_netdev**(int sizeof_priv, const char *name, void (*setup)(struct net_device *));
  - "sizeof_priv": size of private data for driver
  - "name": name of device as seen by user-space. Should have a "%d" in string so that kernel can replace it with the ID of the interface. Ex.: eth0, eth1, etc.
  - "setup": callback to set up rest of struct net_device
- Helper:
  - struct net_device ***alloc_etherdev**(int sizeop_priv);
  - Allocates "eth%d"
  - Provide canned setup function: ether_setup
  - Other such helpers for other types of physical interfaces

- Once device is allocate *and* initialized:
  - int **register_netdevice**(struct net_device *dev);
  - Once this is called, any of the callbacks provided may be invoked.
- Initializing each device:
  - struct net_device cannot be initialized at build time
  - struct net_device is very complex
  - Kernel provides default entries for Ethernet
  - Must set up callbacks, flags, and default values in struct net_device.

- Accessing private data part of struct net_device:
  - struct snull_priv *priv = netdev_priv(dev);
- Module unloading:
  - int **unregister_netdevice**(struct net_device *dev);
  - void **free_netdev**(struct net_device *dev);

3. The net_device structure in detail

- Global information:
  - char name[IFNAMSIZ];
    - Device name
  - unsigned long state;
    - Device state. Use of utility functions to manipulate field
  - struct net_device *next;
    - Next device in global device list
  - int (***init**)(struct net_device *dev);
    - Initialization function
    - Not used by most modern drivers

- Hardware information:
  - unsigned long rmem_end;
    - Receive memory end
  - unsigned long rmem_start;
    - Receive memory start
  - unsigned long mem_end;
    - Transmit memory end
  - unsigned long mem_start;
    - Transmit memory start
  - unsigned long base_addr;
    - I/O base address

    - Not used by the kernel
  - unsigned char irq;
    - Device's IRQ number
  - unsigned char if_port;
    - If multiport device, port in use
  - unsigned char dma;
    - DMA channel
- Helper functions for setting up interface information:
  - void **ether_setup**(struct net_device *dev);
    - Set up for Ethernet device

- void **ltalk_setup**(struct net_device *dev);
  - Set up for LocalTalk device
- void **fc_setup**(struct net_device *dev);
  - Set up for fiber-channel device
- void **fddi_setup**(struct net_device *dev);
  - Set up for FDDI device
- void **hippi_setup**(struct net_device *dev);
  - Set up for HIPPI device
- void **tr_setup**(struct net_device *dev);
  - Set up for token ring device

- Interface information:
  - Usually no need to manipulate by hand unless no helper function is available for the type of adapter being used.
  - unsigned short hard_header_len;
    - Hardware header length
  - unsigned mtu;
    - Maximum Transfer Unit (largest packet size)
  - unsigned long tx_queue_len;
    - Maximum number of packets on transmission queue
  - unsigned short type;
    - Hardware interface type

- unsigned char addr_len;
  - Length of hardware address
- unsigned char broadcast[MAX_ADDR_LEN];
  - Broadcast address on physical network
- unsigned char dev_addr[MAX_ADDR_LEN];
  - Actual device's hardware address
- unsigned short flags;
  - Interface flags bitmask
- int features;
  - Interface features

- Interface flags:
  - IFF_UP:
    - Interface is up
  - IFF_BROADCAST:
    - Interface can broadcast
  - IFF_DEBUG:
    - Print debug info for driver
  - IFF_LOOPBACK:
    - Device is loopback
  - IFF_POINTOPOINT:
    - Device is point-to-point link
  - IFF_NOARP:
    - ARP not supported

- IFF_PROMISC:
  - Device is promiscuous
- IFF_MULTICAST:
  - Device can multicast
- IFF_ALLMULTI:
  - Receive all multicasts
- IFF_MASTER, IFF_SLAVE:
  - Used by load equalization code
- IFF_PORTSEL, IFF_AUTOMEDIA:
  - Set if device can switch between media types
  - Not used by kernel
- IFF_DYNAMIC:
  - Address can change
  - Not used by kernel

- IFF_RUNNING:
  - BSD compatibility
  - Not used by kernel
- IFF_NOTRAILERS:
  - BSD compatibility
  - Not used by kernel

- Features:
  - Set by driver to tell kernel what device can do
  - NETIF_F_SG, NETIF_F_FRAGLIST:
    - Scatter/gather I/O
  - NETIF_F_IP_CSUM, NETIF_F_NO_CSUM, NETIF_F_HW_CSUM:
    - Control whether kernel has to do checksuming or whether device does it.

- NETIF_F_HIGHDMA:
  - High memory DMA-capable
- NETIF_F_HW_VLAN_TX, NETIF_F_HW_VLAN_RX, NETIF_F_HW_VLAN_FILTER, NETIF_F_VLAN_CHALLENGED:
  - Support for 802.1q VLAN support
- NETIF_F_TSO:
  - TCP segmentation offloading

- Fundamental device methods:
  - int (***open**)(struct net_device *dev);
    - Called when device is ifconfig'ed up
  - int (***stop**)(struct net_device *dev);
    - Called when device is ifconfig'ed down

- int (*__hard_start_xmit__)(struct sk_buff *skb, struct net_device *dev);
  - Transmit callback
  - retval is zero on success
  - Non-zero retval will result in kernel retry
- int (*__hard_header__)(struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);
  - Hardware header building callback
- int (*__rebuild_header__)(struct sk_buff *skb);
  - Callback for rebuilding hardware header after ARP completion.

- void (*__tx_timeout__)(struct net_device *dev);
  - Timeout callback
- struct net_device_stats *(*__get_stats__)(struct net_device *dev);
  - Statistics callback
- int (*__set_config__)(struct net_device *dev, struct ifmap *map);
  - Interface config change callback
  - Not typically needed

- Optional device methods:

- int weight; int (*__poll__)(struct net_device *dev, int *quota);
  - NAPI driver poll callback
- void (*__poll_controller__)(struct net_device *dev);
  - Check whether events occured on internface
- int (*__do_ioctl__)(struct net_device *dev, struct ifreq *ifr, int cmd);
  - Device-specific ioctl
- void (*__set_multicast__)(struct net_device *dev);
  - Mutlicast list change callback

- int (*__set_mac_address__)(struct net_device *dev, void *addr);
  - If supported, allow changing MAC address
- int (*__change_mtu__)(struct net_device *dev, int new_mtu);
  - Change MTU
- int (*__header_cache__)(struct neighbour *neigh, struct hh_cache *hh);
  - Fill "hh" with ARP query result
- int (*__header_cache_update__)(struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);
  - Update "hh" cache

- int (*__hard_header_parse__)(struct sk_buff *skb, unsigned char *hadd);
  - Parse hardware headers
- Utility fields:
  - Maintained by driver
  - unsigned long trans_start;
    - Time at which last transmit began, in jiffies
    - Used by networking subsystem to detect lockup
  - unsigned long last_rx;
    - Last receive in jiffies
    - Not currently used

- int watchdog_timeo;
  - Minimum delay for detecting transmission timeout in jiffies.
- void *priv;
  - Driver private data
- struct dev_mc_list *mc_list; int mc_count;
  - Multicast-related fields
- spinlock_t xmit_lock; int xmit_lock_owner;
  - Locks to avoid concurrent access to driver's hard_start_xmit().
  - Should not be modified by driver

4. Opening and closing

- All done via ifconfig
- Bringing up interface with ifconfig:
  - Assign address to interface:
    - ioctl(SIOCSIFADDR)
    - Socket I/O Control Set Interface Address
    - Not seen by driver
    - Handled by kernel
  - Turn interface on:
    - ioctl(SIOCSIFFLAGS)
    - Socket I/O Control Set Interface Flags
    - Results in IFF_UP being set
    - Results in driver's open() callback being invoked

- Driver's open should do:
  - Retrieve MAC address from device
  - Set MAC address in struct net_device (dev->dev_addr).
  - Start up device queue:
    - void **netif_start_queue**(struct net_device *dev);
- Driver's close should do:
  - Stop device queue:
    - void **netif_stop_queue**(struct net_device *dev);

5. Packet transmission

- Basics:
  - Kernel signals transmission to driver using hard_start_xmit().
  - Packets are handed over to driver in form of struct sk_buff (skb).
  - skb contains everything that is needed for packet to travel on network.
  - Driver should usually just transmit skb as-is
  - When transmitted packet shorter than minimum size supported by device, zero-out remainder to avoid security leaks.
  - hard_start_xmit should free skb after transmission

- Controlling transmission concurrency:
  - Calls to hard_start_xmit are serialized using spinlock.
  - Though hard_start_xmit will have initiated a transfer, said transfer will likely not be over at function return.
  - Since devices have limited buffers, driver must tell kernel when no more transfers can be accepted for some time.
  - Use netif_stop_queue to suspend transmission

- Notify kernel that more transfers can start occurring again:
  - void **netif_wake_queue**(struct net_device *dev);
  - Like netif_start_queue, but notifies networking subsystem too.
- If transmission must be suspend elsewhere than in hard_start_xmit():
  - void **netif_tx_disable**(struct net_device *dev);
  - Upon return, means that hard_start_xmit is not running on any CPU.
- Restarting queue after netif_tx_disable() using netif_wake_queue().

- Transmission timeouts:
  - No need to set up timers
  - Handled by network subsystem
  - Must set watchdog_timeo and trans_start
  - If period exceeded, tx_timeout() called automatically
- Scatter/gather I/O:
  - If supported, allows avoiding packet assembly
  - Allows zero-copy
  - NETIF_F_SG must be set for hard_start_xmit() to received fragmented packets.

- Checking whether packet is fragmented:
  - skb_shinfo(skb)->nr_frags
  - If nr_frags != 0, fragmented packet
- In a fragmented skb:
  - ->data points to begining of first fragment
  - First fragment size is: skb->len - skb->data_len
  - Rest of fragments are in ->frags array
  - Each frag is:

```
struct skb_frag_struct {
        struct page *page;
        __u16 page_offset;
        __u16 size;
};
```

  - Must use DMA operations to map struct page as seen earlier.

6. Packet reception

- Need to allocate skb to pass to higher layers
- Reception mode:
  - Interrupt-driven:
    - One interrupt per packet
  - Polled: (high bandwidth)
    - System polls interface for new packets
- Allocate skb:
  - struct sk_buff ***dev_alloc_skb**(unsigned int length);
  - Call is atomic (we're likely in an interrupt handler)

- Set information about packet:
  - Protocol
  - Checksum requirements:
    - CHECKSUM_HW, CHECKSUM_NONE, CHECKSUM_UNNECESSARY
- Set statistics
- Push skb to network stack:
  - int **netif_rx**(struct sk_buff *skb);
  - retval indicates networking subsystem congestion level.
  - Most drivers don't check retval

- High-performance drivers should check retval
- Possible optimization:
  - Preallocate DMA'able skbs
  - Instruct device to transfer directly to those skbs

7. The interrupt handler

- Possible causes:
  - Link status change
  - Transmission complete
  - New packet
- On receipt, do as described in previous section
- On send, deallocate transmitted buffer:
  - **dev_kfree_skb**(struct sk_buff *skb);
  - **dev_kfree_skb_irq**(struct sk_buff *skb);
  - **dev_kfree_skb_any**(struct sk_buff *skb);

8. Receive interrupt mitigation

- Too many packets coming in too rapidly
- Too many interrupts generated
- Kernel provides NAPI: New API
- NAPI does polling on device
- See LDD3 for full details

9. Changes in link state

- Link state may change
- Changing carrier state:
  - void **netif_carrier_off**(struct net_device *dev);
  - void **netif_carrier_on**(struct net_device *dev);
- Checking carrier state:
  - int **netif_carrier_ok**(struct net_device *dev);

10. The socket buffers

- Basic:
  - Unit for packet travel in kernel
- The important fields:
  - struct sk_buff: <linux/skbuff.h>
  - struct net_device *dev;
    - Device responsible for buffer
  - union { ... } h; union { ... } nh; union { ... } mac;
    - Packet header pointers
  - unsigned char *head; unsigned char *data; unsigned char *tail; unsigned char *end;
    - Packet data pointers

- unsigned int len;
  - Actual packet length
- unsigned int data_len;
  - Fragmented packet portion length if scatter/gather
- unsigned char ip_summed;
  - Checksum policy for incoming packet
- unsigned char pkt_type;
  - Packet type
- Shared info struct handling
  - Some info stored in "shared info" struct for performance reasons.

- shinfo(struct sk_buff *skb);
- unsigned int shinfo(skb)->nr_frags;
- skb_frag_t shinfo(skb)->frags;

- Functions acting on socket buffers:
  - struct sk_buff ***alloc_skb**(unsigned int size, int priority);
    - Allocate buffer
  - struct sk_buff ***dev_alloc_skb**(unsigned int length);
    - Allocate buffer with GFP_ATOMIC priority
  - void **kfree_skb**(struct sk_buff *skb);
    - Kernel internal skb freeing

- void **dev_kfree_skb**(struct skb_buff *skb);
  - Driver free skb
- void **dev_kfree_skb_irq**(struct skb_buff *skb);
  - Driver free skb in interrupt handler
- void **dev_kfree_skb_any**(struc skb_buff *skb);
  - Driver free skb in any context
- unsigned char ***skb_put**(struct sk_buff *skb, unsigned int len);
  - Add len to end of buffer
  - Check if enough space

- unsigned char *__**skb_put**(struct sk_buff *skb, unsigned int len);
  - Add len to end of buffer
  - Don't check for space
- unsigned char ***skb_push**(struct sk_buff *skb, unsigned int len);
  - Add len to begining of buffer
  - Used for adding headers
  - Check if enough space
- unsigned char *__**skb_push**(struct sk_buff *skb, unsigned int len);
  - Add len to begining of buffer

- Don't check for space
- int **skb_tailroom**(const struct sk_buff *skb);
  - How much space available for adding data at end of buffer
- int **skb_headroom**(const struct sk_buff *skb);
  - How much space available for adding data at begining of buffer.
- void **skb_reserve**(struct sk_buff *skb, unsigned int len);
  - Reserve "len" bytes at begining and end of buffer
- unsigned char ***skb_pull**(struct sk_buff *skb, unsigned int len);
  - Remove packet head

- int **skb_is_nonlinear**(const struct sk_buff *skb);
  - True if scatter/gather
- unsigned int **skb_headlen**(const struct sk_buff *skb);
  - For s/g, what's the first segment's size
- void ***kmap_skb_frag**(const skb_frag_t *frag);
  - Used to map fragmented skb if it must be accessed within kernel.
- void **kunmap_skb_frag**(void *vaddr);
  - Unamp kmap_skb_frag()

11. MAC address resolution

- Using ARP with Ethernet:
  - Managed by kernel
  - No need for Ethernet driver to do anything special
- Overriding ARP:
  - Use of Ethernet abstractions while avoiding ARP
  - See LDD3 for overiding default Ethernet discovery by kernel.
- Non-Ethernet headers:
  - See LDD3

12. Custom ioctl commands

- <linux/sockios.h> contains list of already recognized commands.
- Typically, commands from user-space are taken care of by protocol-specific ioctl() handler.
- ioctl commands not recognized by protocol callback are passed to driver.
- See LDD3 for more on custom ioctl commands

13. statistical information

- get_stats() callback
- See LDD3 for details

14. Multicast

- See LDD3

15. A few other details

- See LDD3
- Media independent interface support
- Ethtool support
- Netpoll:
  - For remote debugging
  - Provide poll_controller() callback

PCI drivers

1. The PCI interface
2. PCI addressing
3. Boot time
4. Configuration registers and initialization
5. MODULE_DEVICE_TABLE
6. Registering a PCI driver
7. Old-style PCI probing
8. Enabling the PCI device
9. Accessing the configuration space

10. Accessing the I/O and memory spaces
11. PCI interrupts

1. The PCI interface

- Most widely used peripheral bus in mainstream computers.
- PCI specification lays out complete bus functionality.
- Arch-independent
- Higher clock rate than the popular ISA
- 32-bit bus
- Peripherals are auto-configured at boot time
- Linux provides abstractions to help drivers access PCI resources and configuration.

2. PCI addressing

- Peripheral identified using:
  - bus number
  - device number
  - function number
- PCI spec allows a host to have 256 buses
- Linux further supports PCI "domains" to have even more buses.
- Each bus can have 32 devices
- Each device can have up to 8 functions
- Each function identified on PCI bus using 16-bit address/key.

- Linux provides pci_dev struct to manipulate PCI devices.
- Hosts can have many buses plugged together using PCI bridges. Bridges are seen as special PCI peripherals.
- PCI layout is tree of buses and devices
- Root bus is PCI bus 0
- Example layout in fig12-1 in LDD3, p.304
- Viewing of PCI layout can be done using "lspci", or looking at /proc/pci, /proc/bus/pci or /sys/bus/pci/.

- Ways in which devices in PCI layout are displayed (in hex):
  - 2 values: 8 bit bus ID, 8 bit device and function number
  - 3 values: bus, device, and function
  - 4 values: domain, bus, device, and function
- Peripherals can answer queries about:
  - Memory locations
  - I/O ports
  - Configuration registers

- All devices share address space for:
  - Memory locations (32 or 64-bit range)
  - I/O ports (32-bit range)
- Configuration registers conform to "geographical addressing" => no share.
- Access to memory locations and I/O ports done using I/O access functions covered earlier.
- Layout of addressable locations done at boot and mapping avoids collisions.

- Access to configuration done through specific kernel functions.
- Interrupts:
  - Each PCI peripheral has 4 interrupts pins and it can use any of them, regardless of the actual routing of the interrupt to the CPU.
  - Interrupts are shared
- Each PCI device function has 256 bytes for config (PCIX has 4K)
  - 4 bytes reserved for unique ID (used by drivers to locate device).

3. Boot time

- Upon reset PCI devices:
  - Have no memory or I/O mapping
  - Remain in quiescent mode
  - Interrupts disabled
- "Firmware" comes up and configures devices via PCI controller, allocating unique I/O and memory ranges.
- Linux can do this too
- Current device configuration table readable through /proc/bus/pci/devices.

- Sysfs provides per-device configuration items through /sys/bus/pci/devices/<a-device>:
  - config: binary PCI config info
  - vendor, device,subsystem_device, subsystem_vendor, class: device-specific info.
  - irq: IRQ assigned to device
  - resource: memory resources currently allocated to device.

4. Configuration registers and initialization

- Device (function) configuration contained in 256 bytes
- See LDD3 p.308 for illustration
- First 64 bytes are standard and required
- The rest depends on peripheral
- All PCI registers are little-endian
- Use proper byte-ordering functions when necessary.
- See peripheral hardware documentation for meaning and use of registers.

- Fields of interest for driver:
  - vendorID (16 bit)
    - Unique vendor identificator
    - Each PCI peripheral manufacturer has a different ID
  - deviceID (16 bit)
    - Vendor-attributed ID
    - No central registry
    - Known in combination with vendorID as device "signature".
  - class (16 bit)
    - Top 8 bit is "base class" or group:
      - "network" group contains: Ethernet and token ring

        - "communication" group contains: serial and parallel
      - Existing classes are defined in PCI spec
    - subsystem vendorID, subsystem deviceID:
      - Sometimes many PCI boards will be based on the same basic PCI interface chip. These IDs would be used to identify which one of the actual boards this one is.
- Drivers use these fields to indicate which peripherals they support:
  - struct pci_device_id
  - __u32 vendor; __32 device;
    - PCI vendor and device IDs
    - Use PCI_ANY_ID if support for any

- __u32 subvendor; __32 subdevice;
  - PCI subsystem vendor and subsystem device IDs
  - Use PCI_ANY_ID if support for any
- __u32 class; __u32 class_mask;
  - PCI class
  - Use PCI_ANY_ID if support for any
- kernel_ulong_t driver_data;
  - Device-specific data if driver supports more than one device.
- Helper macros for creating pci_device_id structs:

- PCI_DEVICE(vendor, device):
  - Creates struct pci_device_id of "vendor" and "device"
  - Set subsystem vendor and subsystem device ID to PCI_ANY_ID.
- PCI_DEVICE_CLASS(device_class, device_class_mask)
  - Creates struct pci_device_id matching specific class
- Can declare list of struct pci_device_id to give to PCI layer to indicate the list of supported devices.

5. MODULE_DEVICE_TABLE

- In order for user-space hotplug functionality to know which devices are supported by each driver, the list of supported devices must be exported by driver modules.
- MODULE_DEVICE_TABLE(<bus>, <list of struct pci_device_id>);
- For pci, use "pci" for <bus>
- Module will now have a __mod_pci_device_table symbol exported.

- "depmod" creates /lib/modules/KERNEL_VERSION/modules.pcimap based on the list of __mod_pci_device_table symbols found in all modules.
- modules.pcimap matches device IDs with module names.
- The hotplug functionality uses modules.pcimap to know which driver to load in case a new PCI device is found.

6. Registering a PCI driver

- struct pci_driver:
  - const char *name;
    - Driver name
    - Must be unique throughout kernel
    - Normally set to module name
    - Shows up in /sys/bus/pci/drivers/
  - const struct pci_device_id *id_table;
    - PCI device ID table
  - int (***probe**) (struct pci_dev *dev, const struct pci_device_id *id);
    - Driver probe callback
    - Called by kernel a struct pci_dev is found for this driver

- "id" is the device the kernel has found for this driver
- retval should be zero if driver accepts responsibility for device and has initialized said device.
- retval should be negative error code if driver doesn't recognize device or doesn't want to handle it.

- void (***remove**) (struct pci_dev *dev);
  - Called when device is being removed from the system
  - Called when PCI driver is being unloaded from the kernel
- int (***suspend**) (struct pci_dev *dev, u32 state);
  - Optional
  - Called when device is getting suspended
  - "state" is suspend state

- int (*__resume__) (struct pci_dev *dev);
  - Optional
  - Called to reverse suspend()
- Basic PCI driver entry:

```
static struct pci_driver my_pci_driver = {
        .name = "my_pci_driver",
        .id_table = my_ids,
        .probe = my_probe,
        .remove = my_remove
}
```

- PCI driver registration:
  - int **pci_register_driver**(struct pci_driver *pci_driver);
  - retval < 0 if error

- retval is zero if success
- Does not return error if no device bound to driver because of:
  - Hotplug
  - Dynamically created ID (i.e. IDs not already recognized by kernel).

- PCI driver removal:
  - void **pci_unregister_driver**(struct pci_driver *pci_driver);
  - Results in calls to remove() for every device PCI device bound to this driver.
  - Function doesn't return until all remove() calls return

7. Old-style PCI probing

- Manual PCI list probing
- Can't call these functions from interrupt handlers
- struct pci_dev ***pci_get_device**(unsigned int vendor, unsigned int device, struct pci_dev *from);
  - Looks for PCI device in list of existing devices
  - Increments ref count on found pci_dev struct
  - First call should have "from" set to NULL
  - Subsequent calls should pass the device already returned to start looking for more devices "after" that one.

  - When no more devices, function returns NULL
- void **pci_dev_put**(struct pci_dev *pci_dev);
  - Decrement ref count on device
- struct pci_dev ***pci_get_subsys**(unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
  - Same as above, but allows passing subsystem vendor and subsystem device.
- struct pci_dev ***pci_get_slot**(struct pci_bus *bus, unsigned int devfn);

- Searches a specific bus for a given device function

8. Enabling the PCI device

- Upon having its probe() function called, a driver must tell the kernel to "enable" the device if it intends to use it:
  - int **pci_enable_device**(struct pci_dev *dev);
    - "Wakes up" device
    - In some cases, assigns interrupt line and I/O regions

9. Accessing the configuration space

- Having found device, driver may need to read and/or write to
  - Memory space
  - I/O port space
  - Configuration space (essential for knowing where device is located)
- Accessing configuration space based on PCI controller chip implementation.
- Linux provides controller-independent functions:
  - Must provide distance from begining of config space "where" to read from in bytes.

- Functions return error code
- Functions reading more than one byte automatically convert data from or to little-endian, to or from the local processor's byte ordering.

- <linux/pci.h>
- Reading config data (8-, 16- and 32-bit):
  - int **pci_read_config_byte**(struct pci_dev *dev, int where, u8 *val);
  - int **pci_read_config_word**(struct pci_dev *dev, int where, u16 *val);

  - int **pci_read_config_dword**(struct pci_dev *dev, int where, u32 *val);
- Writing config data:
  - int **pci_write_config_byte**(struct pci_dev *dev, int where, u8 val);
  - int **pci_write_config_word**(struct pci_dev *dev, int where, u16 val);
  - int **pci_write_config_dword**(struct pci_dev *dev, int where, u32 val);
- Previous functions are actually macros

- Low-level operations:
  - int **pci_bus_read_config_byte**(struct pci_bus *bus, unsigned int devfn, int where, u8 *val);
  - int **pci_bus_read_config_word**(struct pci_bus *bus, unsigned int devfn, int where, u16 *val);
  - int **pci_bus_read_config_dword**(struct pci_bus *bus, unsigned int devfn, int where, u32 *val);
  - int **pci_bus_write_config_byte**(struct pci_bus *bus, unsigned int devfn, int where, u8 val);
  - int **pci_bus_write_config_word**(struct pci_bus *bus, unsigned int devfn, int where, u16 val);

- int **pci_bus_write_config_dword**(struct pci_bus *bus, unsigned int devfn, int where, u32 val);

- Predefined locations "where" to read from in <linux/pci.h>:
  - PCI_VENDOR_ID
  - PCI_DEVICE_ID
  - PCI_COMMAND
  - PCI_STATUS
  - PCI_REVISION_ID
  - PCI_INTERRUPT_LINE
  - etc.

10. Accessing the I/O and memory spaces

- PCI devices have up to 6 I/O address regions
- Each region has either memory or I/O
- Most devices have memory-mapped registers
- Remember, device registers should not be cached.
- Peripherals indicate whether they want certain regions to be prefetchable or not (cached or not.)
- Peripherals report address regions in 256-byte config structure.

- Accessed using config-access functions at locations:
  - PCI_BASE_ADDRESS_0
  - PCI_BASE_ADDRESS_1
  - ...
  - PCI_BASE_ADDRESS_5
- These registers used to define both 32-bit memory regions and 32-bit I/O regions.
- 64-bit memory regions can be declared using 2 consecutive config registers.

- Use kernel helper functions instead of accessing config directly:
  - unsigned long **pci_resource_start**(struct pci_dev *dev, int bar);
    - retval is first address of given memory region
    - "bar" indicates for which Base Address Register the start location is sought (from 0-5).
  - unsigned long **pci_resource_end**(struct pci_dev *dev, int bar);
    - retval is last usable address of given region

- unsigned long **pci_resource_flags**(struct pci_dev *dev, int bar);
  - retval is given region's flags
- Region flags:
  - <linux/ioport.h>
  - IORESOURCE_IO:
    - Region exists
    - Region is I/O
  - IORESOURCE_MEM:
    - Region exists
    - Region is memory

- IORESOURCE_PREFETCH:
  - Region can be prefetched
- IORESOURCE_READONLY:
  - Region is read-only
  - Never happens for PCI devices
- Use previously-discussed I/O functions to read/write into PCI regions.

11. PCI interrupts

- Usually attributed as part of PCI POST at boot
- Just a matter of requesting which PCI interrupt has been attributed to given PCI device:

```
retval = pci_read_config_byte(my_dev, PCI_INTERRUPT_LINE, &my_irq);
```

- After that, use the already-covered request_irq(), etc.

USB drivers

1. USB device basics
2. USB and sysfs
3. USB urbs
4. Writing a USB driver
5. USB transfers without urbs

1. USB device basics

- Properties:
  - Tree of devices
  - Single-master
  - Low-cost
  - No data format enforced
  - Can preassign bandwidth
  - Some device types already defined (classes). As long as device behaves as in the standard, a generic driver for that device type can be used as-is.

- Linux and USB:
  - Linux supports USB as host and as device (gadget API)
  - We will not cover gadget API, see doc on web
  - Linux provides USB core stack for allowing drivers to interface with USB hardware, via the USB host controllers, transparently.
  - In Linux, devices consist of:
    - Config
    - Interface (contained in config)
    - Endpoint (contained in interface)
  - Linux USB drivers attach to "interface", not entire device.

- Endpoints:
  - Basic form of USB communication
  - Can carry data in one direction only
    - OUT: From computer to device
    - IN: From device to computer
  - endpoint "size": amount of data that can be held by an endpoint at once.
  - Types of endpoints:
    - CONTROL:
      - Asynchronous transfers: Used when needed
      - Transfer small amounts of data
      - Read/write config info

    - Write commands
    - Read status
    - Each device has at least "endpoint 0"
    - At insertion time, USB core uses "endpoint 0" to config device
    - "endpoint 0" transfers guaranteed by USB protocol
- INTERRUPT:
    - Period transfers: Bandwidth reserved
    - Transfer small amounts of data at fixed rate from device to computer when host asks.
    - Primary transport for mice and keyboards
    - Can also be used to send commands to devices
    - Transfers guaranteed by USB protocol
- BULK:
    - Asynchronous transfers
    - Transfer large amounts of data
    - Lossless transfers

    - No guarantee on latency
    - Transfers may be broken-down into smaller-sized transfers
    - Commonly used by devices such as printers, storage, and network
  - ISOCHRONOUS:
    - Period transfers
    - Transfer large amounts of data
    - No guarantee data makes it through
    - Best for streaming devices that can loose some data
    - Commonly used for data acquisition, audio, vidoe.
- Linux struct for endpoints:
  - struct usb_host_endpoint
  - Contains actual endpoint information placeholder:
    - struct usb_endpoint_descriptor
    - Data in this struct is as passed by device

- Placeholder entries relevant to drivers:
  - bEndpointAddress:
    - Endpoint's USB address
    - Use USB_DIR_OUT and USB_DIR_IN bitmasks to determine direction.
  - bmAttributes:
    - Endpoint type
    - Use USB_ENDPOINT_XFRERTYPE_MASK, USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BUL, or USB_ENDPOINT_XFER_INT to determine endpoint type.
  - wMaxPacketSize:
    - Maximum packet size handled by endpoint at every transfer
    - Larger transfers will be cut into wMaxPacketSize
    - See the USB spec to use this field to specify a "high-bandwidth" mode.
  - bInterval
    - Interval in milliseconds for interrupt-type transfers

- USB naming in Linux conforms to USB spec, not Linux standard, hence the sometimes odd-named variables.

- Interfaces:
  - Endpoint bundle (zero or more)
  - An interface represents at most one logical USB connection.
  - Drivers are tied to single interfaces
  - Devices that present more than one connection (a keyboard with acceleration keys, for example) must use more than one driver.

- Interfaces can often be used in different "configurations": Alternate settings.
- Each configuration represents a set of device settings, possibly reserving different bandwidths depending on operating mode.
- Initial device setting known as first setting (numbered as 0).
- Linux struct for interfaces (passed to and used by USB drivers):
  - struct usb_interface
  - Important fields:
    - struct usb_host_interface *altsetting:
      - Array of possible alternate settings for interface

        - Each usb_host_interface corresponds to set of endpoint configs (struct usb_host_endpoint).
        - No particular ordering
    - unsigned num_altsetting:
        - Number of alternate settings
    - struct usb_host_interface *cur_altsetting:
        - Currently active alternate setting
        - Pointer into altsetting
    - int minor:
        - All USB devices have the same major number
        - This is the minor number attributed to device after call to usb_register_dev().
- Configurations:
    - Interface bundle (one or many)

- Linux struct for configurations:
  - struct usb_host_config
- Device:
  - Usually one config
  - Some device scan have multiple configurations (rare).
  - Only one configuration can be active for a given device at one time.
  - Support for multiple-configuration devices in Linux is poor.
  - Linux struct for entire device: struct usb_device

- Converting data from struct usb_interface to struct usb_device:
  - interface_to_usb() macro
  - Common operation for drivers

2. USB and sysfs

- Sysfs represents both USB device and USB interfaces as individual devices.
- Sysfs USB hierarchy largely depends on kernel labeling of USB devices.
- Usually, the root entry for USB devices in sysfs is where the USB controller is located on the PCI bus. Example:

  /sys/devices/pci0000:00/0000:00:1d.0/usb2/

- First device in USB tree is USB root hub
- Each USB root hub has unique ID (here this is 2)

- No limit on number USB root hubs
- Sysfs naming scheme for USB device directly connected to root hub:

  <root hub>-<hub port>:<config nb>.<interface>
- Device connected using external USB hub:

  <root hub>-<hub port>-<hub port>:<config nb>.<interface>
- Most information about USB device is available in its sysfs entry, often using the same naming scheme as in spec. Examples:

  - bConfigurationValue
  - bDeviceClass
  - serial
  - etc.
- bConfigurationValue can be written to in order to set which config device should use.
- Standard sysfs doesn't provide internal device info
- More in-depth information about USB devices can be found usbfs, which is usually mounted at /proc/bus/usb.

- /proc/bus/usb/devices is of special interest as it exposes the alternate configuration info along with info regarding endpoints.
- User-space USB drivers such as the one for scanners can talk directly to USB hardware via usbfs.

3. USB urbs

- Basics:
  - Communication to and from endpoints done through USB Request Blocks (URBs) in Linux.
  - struct urb: <linux/usb.h>
  - A single urb can be used on many endpoints
  - A single endpoint can have many urbs allocated for it.
  - Each endpoint has urb queue
  - urbs are usually queued for processing
  - Queued urbs can be canceled (by either driver or USB core).

- Contain internal ref count (deleted on free of last ref)
- urb lifecycle:
  - USB driver creates it
  - Is assigned to single endpoint
  - Submitted by driver to USB core
  - Submitted by USB core to USB controller
  - Processed by USB controller for transfer with device
  - USB controller notifies driver of urb completion
- struct urb important fields:
  - struct usb_device *dev:
    - USB device to which urb belongs to

    - Must set prior to urb queuing
- unsigned int pipe:
    - Type of endpoint (only one type possible)
    - Set using endpoint setting functions
    - Must set prior to urb queuing
- unsigned int transfer_flags:
    - Specify how urb should be dealt with
    - Predefined urb transfer flags
- void *transfer_buffer:
    - Buffer to transmit to or from
    - Use kmalloc'ed buffers only

- dma_addr_t transfer_dma:
  - Used instead of transfer_buffer if buffer is DMA
- int transfer_buffer_length:
  - Length of transfer_buffer or transfer_dma
  - If larger than maximum allowed by endpoint, transfers will be broken up into separate USB frames.
  - Best to let USB core take care of breaking up transfers instead of conducting short transfers.
- unsigned char *setup_packet:
  - For control urbs only
  - Buffer sent before control data is transferred

- dma_addr_t setup_dma:
  - For control urbs only
  - Used instead of setup_packet if buffer is DMA
- usb_complete_t complete:
  - Callback invoked when urb transfer is completed or in case of error.
- void *context:
  - Private data that could be used by completion callback
- int actual_length:
  - Actual size of transfer
  - Use this instead of transfer_buffer_length on input

- int status:
  - Current urb status
  - List below
  - Can be accessed safely in completion callback only
  - Use isochronous endpoints, use iso_frame_desc[] for real status.
- int start_frame:
  - For isochronous transfers only
  - First frame in or out
- int interval
  - For isochronous or interrupt urbs only
  - Must be sent prior to urb enqueued

  - Interval for urb polling
  - Units depend on device speed
- int number_of_packets:
  - For isochronous urbs only
  - Number of isochronous packets to be handled by urb
  - Must be set prior to urb enqueing
- int error_count:
  - For isochronous urbs only
  - Number of errors during isochronous urb transfer
- struct usb_iso_packet_descriptor iso_frame_desc[0]:
  - For isochronous urbs only
  - Array of struct usb_iso_packet_descriptor

- Defines or collect status of a number of isochronous transfers.
- struct usb_iso_packet_descriptor:
  - unsigned int offset:
    - Offset in transfer buffer for this packet's content
  - unsigned int length:
    - Length for packet in transfer buffer
  - unsigned int actual_length:
    - Actual length received
  - unsigned int status:
    - Transfer status for this particular packet
    - Same definitions as for urb's "status"

- Setting urb endpoint type:
  - unsigned int **usb_sndctrlpipe**(struct usb_device *dev, unsigned int endpoint);

- unsigned int **usb_rcvctrlpipe**(struct usb_device *dev, unsigned int endpoint);
- unsigned int **usb_sndbulkpipe**(struct usb_device *dev, unsigned int endpoint);
- unsigned int **usb_rcvbulkpipe**(struct usb_device *dev, unsigned int endpoint);
- unsigned int **usb_sndintpipe**(struct usb_device *dev, unsigned int endpoint);
- unsigned int **usb_rcvintpipe**(struct usb_device *dev, unsigned int endpoint);
- unsigned int **usb_sndisocpipe**(struct usb_device *dev, unsigned int endpoint);

- unsigned int **usb_rcvisocpipe**(struct usb_device *dev, unsigned int endpoint);

- urb flags:
  - URB_SHORT_NOT_OK:
    - Flags short reads on IN endpoint as error
  - URB_ISO_ASAP:
    - Schedule isochornous urb ASAP, as soon as bandwidth allows it.
    - Drivers must be able to deal with non-imminent transfer if bit is not set for isochronous transfer.
  - URB_NO_TRANSFER_DMA_MAP:
    - Set to tell USB core buffer transfered is DMA

    - USB core uses transfer_dma instead of transfer_buffer
- URB_NO_SETUP_DMA_MAP:
    - Similar to previous
    - USB core uses setup_dma instead of setup_packet
- URB_ASYNC_UNLINK:
    - If set, urb unlinking will happen in the background instead of the urb unlinking function blocking until urb is truly unlinked.
    - Use with care => complicated debugging
- URB_NO_FSBR:
    - Check LDD3, for UHCI USB host controller driver only
- URB_ZERO_PACKET:

        - Bulk out transfer followed by short packet containing no data.
        - Required for some broken peripherals
    - URB_NO_INTERRUPT:
        - Don't generate interrupt after dealing with urb
        - Use with great care
- urb status:
    - 0:
        - Transfer successful
    - ENOENT:
        - urb_kill_urb() stopped urb

- ECONNRESET:
  - urb_unlink_urb() with transfer_flags set to URB_ASYNC_UNLINK.
- EINPROGRESS:
  - Host controller still processing urb
  - Buggy driver
- EPROTO:
  - (?) bitstuff error
  - Hardware didn't received response packet in time
- EILSEQ:
  - CRC error

- EPIPE:
  - Endpoint stalled
  - If not control endpoint, clear using usb_clear_halt()
- ECOMM:
  - Incoming transfer was too fast
  - Couldn't write input to memory
- ENOSR:
  - Outgoing transfer was too fast
  - Couldn't retrieve output fast enough
- EOVERFLOW:
  - Receive more data than maximum size set for endpoint

- EREMOTEIO:
  - Having set URB_SHORT_NOT_OK, less data was received than expected.
- ENODEV:
  - Device not connected anymore
- EXDEV:
  - For isochronous transfers only
  - Transfer was incomplete
  - Look at iso_fram_desc for understanding where transfer failed.
- EINVAL:
  - Critical error with urb

- ESHUTDOWN:
  - Device has serious problem, system shut it down
- Creating and destroying urbs:
  - Should always dynamically create urbs to preserve ref counting.
  - Basic urb creation
    - struct urb ***usb_alloc_urb**(int iso_packets, int mem_flags);
      - "iso_packets": number of isochronous packets, set to 0 if not isochronous urb.
      - "mem_flags": same as kmalloc() flags
      - retval non-NULL is success
      - retval NULL on failure
      - urb must be properly initialized prior to enqueing

    - void **usb_free_urb**(struct urb *urb);
        - Frees urb
- Initialization helper functions do not set transfer_flags, must be done by driver if needed after urb initialization.
- Interrupt urbs:
    - void **usb_fill_int_urb**(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer, int buffer_length, usb_complet_t complete, void *context, int interval);
    - "urb": urb to be initialized
    - "dev": device where to send urb

- "pipe": endpoint type, see earlier functions:
  - usb_sndintpipe / usb_rcvintpipe
- "transfer_buffer": kmalloc'ed intput/output buffer
- "buffer_length": length of transfer_buffer
- "compete": completion callback
- "context": private data for completion callback
- "interval": urb scheduling interval

- Bulk urbs:
  - void **usb_fill_bulk_urb**(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer, int buffer_length, usb_complet_t complete, void *context);
  - Paramaters same as usb_fill_int_urb();
  - "pipe": usb_sndbulkpipe or usb_rcvbulkpipe

- Control urbs:
  - void **usb_fill_control_urb**(struct urb *urb, struct usb_device *dev, unsigned int pipe, unsigned char *setup_packet, void *transfer_buffer, int buffer_length, usb_complet_t complete, void *context);
  - Parameters same as usb_fill_bulk_urb()
  - "setup_packet": send prior to data
  - "pipe": usb_sndctrlpipe or usb_rcvctrlpipe
  - This initializer not typically used
  - Direct transfers are used instead (will see later)
- Isochronous urbs:
  - No helper functions
  - Must be initialized by hand

    - See LDD3 p.344 for example
- Submitting urbs:
  - int **usb_submit_urb**(struct urb *urb, int mem_flags);
  - Can be called within interrupt context
  - "urb": urb to send
  - "mem_flags": same as kmalloc flags; indicates how USB core should allocate any required memory. Valid values:
    - GFP_ATOMIC: critical situation, such as int handler, where sleeping cannot be allowed.
    - GFP_NOIO: driver is doing block I/O

    - GFP_KERNEL: most situations
  - retval is 0 on success
  - retval < 0 on error
  - Must wait for completion flag before accessing any urb fields.
- Completing urbs: the completion callback handler
  - typedef void (***usb_complete_t**)(struct urb *, struct pt_regs *);
  - Called only once for each urb completion
  - Likely running in interrupt context

- Possible reasons for completion:
  - urb sent successfully and device acks appropriately
  - Error happened during transfer
  - urb unlinked
- Canceling urbs:
  - int **usb_kill_urb**(struct urb *urb);
    - Typically used device disconnect callback
  - int **usb_unlink_urb**(struct urb *urb);
    - Must have set URB_ASYNC_UNLINK to work properly
    - Will not block while urb is being unlinked

4. Writing a USB driver

- What devices does the driver support?
  - struct usb_device_id (much like pci_device_id)
  - Used by USB to determine which devices a driver can handle.
  - Used by hotplug scripts to determine which drivers to load.
  - __u16 match_flags:
    - USB_DEVICE_ID_MATCH* from <linux/mod_devicetableh.h>.
    - Never set directly
    - Set using USB_DEVICE() macro

- __u16 idVendor:
  - Unique vendor ID
- __u16 idProduct:
  - Vendor-specific product ID
- __u16 bcdDevice_lo, __u16 bcdDevice_hi:
  - Supported product version range
  - Specified in Binary-Coded Decimal (BCD)
- __u8 bDeviceClass, __u8 bDeviceSubClass, __u8 bDeviceProtocol:
  - Define class, subclass and protocol as spelled out by spec
  - Values specify device behavior

- __u8 bInterfaceClass, __u8 bInterfaceSubClass, __u8 bInterfaceProtocol
  - Like previous
  - Values specify interface
- kernel_ulong_t driver_info:
  - Set by kernel to identify different devices when probing
- Helper macros:
  - USB_DEVICE(vendor, product):
    - Very commonly used
    - Create struct usb_device_id matching vendor and product IDs
  - USB_DEVICE_VER(vendor, product, lo, hi):
    - Like previous, but set supported product versions

- USB_DEVICE_INFO(class, subclass, protocol):
  - Create struct usb_device_id matching class description
- USB_INTERFACE_INFO(class, subclass, protocol):
  - Create struct usb_device_id matchin interface description
- MODULE_DEVICE_TABLE(usb, <list of struct usb_device_id>);

- Registering a USB driver:
  - struct usb_driver:
    - struct module *owner:
      - Pointer to module owning this driver
      - Ref counting by USB core
      - Usually set to "THIS_MODULE"

- const char *name:
  - Device name
  - Must be unique accross all USB devices in kernel
  - Shows up in /sys/bus/usb/drivers/
- const struct usb_device_id *id_table:
  - Pointer to list of struct usb_device_id supported by driver
- int (***probe**) (struct usb_interface *intf, const struct usb_device_id *id):
  - Called by USB core when match is likely to have been found
  - "id" is the likely match
  - retval should be 0 if claimed and initialized
  - retval should be < 0 if no claim or error
  - Called within the context of khubd kernel thread, legal to sleep
  - Typically:
    - Initialize any local structures/variables for managing device

        - Record any info regarding device in local structs for future use.
- void (*__disconnect__) (struct usb_interface *intf):
    - Invoked on device disconnect
    - Invoked on driver removal
    - Called within the context of khubd kernel thread, legal to sleep
- int (*__ioctl__) (struct usb_interface *intf, unsigned int code, void *buf):
    - Not typically implemented
    - Not typically needed
    - Called when uspace app does ioctl() on usbfs device entry
- int (*__suspend__) (struct usb_interface *intf, u32 state)
    - Not typically implemented
    - Called on device suspend

- int (***resume**) (struct usb_interface *intf)
  - Not typically implemented
  - Called on device resume
- Basic usb_driver entry:

```
static struct usb_driver my_driver = {
        .owner = THIS_MODULE,
        .name = "MyDriver",
        .id_table = my_table,
        .probe = my_probe,
        .disconnect = my_disconnect
};
```

- Actual registration:
  - int **usb_register**(struct usb_driver *);
  - void **usb_deregister**(struct usb_driver *);
- Most important work should be conducted at device open from uspace.

- probe and disconnect in detail
  - See example in LDD3, p.350
  - Linking a struct usb_device_id to a struct usb_interface:
    - void **usb_set_intfdata** (struct usb_interface *intf, void *data);
  - Getting struct usb_device_id from struct usb_interface:
    - void ***usb_get_intfdata** (struct usb_interface *intf);
    - Usually done on disconnect or device open()
  - When disconnecting, do not forget to use usb_set_intfdata() to reset private data to NULL.

- Connection between higher-levels and USB driver:
  - Usually, USB driver interface to uspace using the other subsystem it ties into, like scsi, network, etc.
  - USB drivers can also tie to user-space using a char device interface:
    - int **usb_register_dev**(struct usb_interface *intf, struct usb_class_driver *class_driver);
      - Call in probe()
      - struct usb_class_driver:
        - Parameters for obtaining minor number
        - char *name;
          - sysfs name
          - Use "%d" if many devices

        - struct file_operations *fops;
            - Char fileops
        - mode_t mode;
            - File access mode
        - int minor_base;
            - Base minor number
    - void **usb_deregister_dev**(struct usb_interface *intf, struct usb_class_driver *class_driver);
        - Called in disconnect()
- USB buffer allocation primitives:
    - void ***usb_buffer_alloc** (struct usb_device *dev, size_t size, int mem_flags, dma_addr_t *dma);
        - "dev": usb device
        - "size": amount requested

  - "mem_flags": same as kmalloc()
  - "dma": "transfer_dma" entry in urb struct
- void **usb_buffer_free** (struct usb_device *dev, size_t size, void *addr, dma_addr_t dma);
  - Similar as above
  - "addr" is usb_buffer_alloc'ed space

5. USB transfers without urbs

- Transfer without dealing with urbs at all
- Just send and receive direct USB
- usb_bulk_msg:
  - int **usb_bulk_msg**(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, int *actual_length, int timeout);
  - "usb_dev": the usb device
  - "pipe": endpoint type
  - "data": input/output buffer
  - "len": data buffer len

- "actual_length": ptr to nb of bytes actually transfered
- "timeout": timeout in jiffies, if 0 it'll wait forever
- retval is 0 on success
- retval < 0 on error, see "struct urb" errors for actual error.
- Cannot be called from interrupt
- Call cannot be cancelled
- disconnect() must wait if call issued

- usb_control_msg:
  - int **usb_control_msg**(struct usb_device *usb_dev, unsigned int pipe, __8 request, __8 requesttype, __16 value, __16 index, void *data, __u16 size, int timeout);
  - "usb_dev": the usb device
  - "pipe": endpoint type
  - "request": USB request per spec
  - "requesttype": USB request type per spec
  - "value": USB message value per spec
  - "index": USB message index per spec

- "data": input/output buffer
- "size": size of data
- "timeout": timeout in jiffies
- retval is number of bytes transfer on success
- retval is negative error value on error
- Cannot be called from interrupt
- Call cannot be cancelled
- disconnect() must wait if call issued

- Other USB data functions:
  - Obtain standard info from USB devices

- Cannot call in interrupt handlers
- int **usb_get_descriptor**(struct usb_device *dev, unsigned char type, unsigned char index, void *buf, int size);
  - Can be used to retrieve information not already available throug the USB structures described earlier.
  - "dev": the usb device
  - "type": type according to spec:
    - USB_DT_DEVICE, USB_DT_CONFIG, USB_DT_STRING, USB_DT_INTERFACE, USB_DT_ENDPOINT, USB_DT_DEVICE_QUALIFIER, USB_DT_OTHER_SPEED_CONFIG, USB_DT_INTERFACE_POWER, USB_DT_OTG, USB_DT_DEBUG, USB_DT_INTERFACE_ASSOCIATION, USB_DT_CS_DEVICE, USB_DT_CS_CONFIG, USB_DT_CS_STRING, USB_DT_CS_INTERFACE, USB_DT_CS_ENDPOINT.

  - "index": descriptor number
  - "buf": buffer to copy descriptor to
  - "size": size of "buf"
  - retval is number of bytes transfered on success
  - retval is negative value in case of error.
  - Relies on usb_contro_msg()
- int **usb_get_string**(struct usb_device *dev, unsigned short langid, unsigned char index, void *buf, int size);

TTY drivers

1. Basics
2. A small TTY driver
3. tty_driver function pointers
4. TTY line settings
5. ioctls
6. proc and sysfs handling of TTY devices
7. Core struct details

1. Basics

- tty = TeleTYpewriter
- Any serial-port-like device
- tty virtual devices used to create interactive sessions through various software abstractions (X, network, etc.)
- Types of TTYs:
  - Virtual console:
    - Full-screen terminal displays on the system video monitor
  - System console:
    - The device to which system messsages should be sent, and where logins should be permitted in single-user mode.

- Serial ports:
  - RS-232 serial ports and any device which simulates one, either in hardware (such as internal modems) or in software (such as the ISDN drivers or USB-to-serial drivers.)
- Pseudo-terminals (PTYs):
  - Used to create login sessions or provide other capabilities requiring a TTY line discipline to arbitrary data-generating proccesses.
- What is a "line discipline"?:
  - A conversion layer between the TTY driver which talks to actual hardware and a software layer that knows only how to talk to a generic TTY.

  - Typically, this is a protocol conversion: PPP, Bluetooth, etc.
- See /proc/tty/drivers for list of tty drivers currently loaded.

2. A small TTY driver

- Basics:
  - struct tty_driver: <linux/tty_driver.h>
  - Allocating a tty driver:
    - struct tty_driver ***alloc_tty_driver**(<nb tty devices supported>);
  - After allocation, struct tty_driver should be initialized.
  - May want to set struct tty_operations using tty_set_operations().
  - Once initialized, the driver should be registered:
    - int **tty_register_driver**(struct tty_driver *driver);
    - Results in sysfs entries creation

- Once registered, the driver should register the devices it controls:
  - void **tty_register_device**(struct tty_driver *driver, /* tty driver */ unsigned index, /* Minor nbr */ struct device *dev); /* Device */
  - Conversely:
    - void **tty_unregister_device**(struct tty_driver *driver, unsigned index);
    - int **tty_unregister_driver**(struct tty_driver *driver);
- struct tty_driver contents
  - "owner": THIS_MODULE
  - "driver_name": name shown in /proc/tty/drivers

- "name": entry in /dev
- "major": major number
- "type" and "subtype": TTY driver type
- "flags": driver state and type

- struct termios:
  - struct tty_driver has init_termios (struct termios) member.
  - init_termios are the default line settings for the driver.
  - These may be changed later when device is open from user-space.

- Bitmask entries in struct termios (see termios manpage):
  - tcflag_t c_iflag;
    - Input mode
  - tcflag_t c_oflag;
    - Output mode
  - tcflag_t c_cflag;
    - Control mode
  - tcflag_t c_lflag;
    - Local mode
  - cc_t c_line;
    - Line discipline type
  - cc_t c_cc[NCCS];
    - Control characters array

3. tty_driver function pointers

- open and close:
  - Open called by TTY layer when uspace open on /dev entry.
  - Close called by TTY layer when uspace close on /dev entry.
- Flow of data:
  - Write called as a result of uspace write
  - Write can be called from interrupt context
- Other buffering functions:
  - put_char: add single character to output buffer
  - chars_in_buffer: how many bytes left to send

- flush_chars: start writing chars to device, if not already done.
- wait_until_sent: same as flush_chars, but wait for send to complete.
- flush_buffer: send as much of what remains as possible, and drop what's left.

- No read function?
  - TTY layer buffers input until user reads
  - TTY layer notifies driver if stop/start needed

- Feeding characters to TTY layer:
  - void **tty_insert_flip_char**(struct tty_struct *tty, unsigned char ch, char flag);
- Flushing buffer when enough characters:
  - void **tty_flip_buffer_push**(struct tty_struct *tty);

4. TTY line settings

- Basics:
  - User-space attempts to set TTY config using ioctl()
  - TTY layer recognizes some of the ioctl() calls and converts them to driver callbacks.
- set_termios:
  - Most common callback to TTY ioctls
  - Parity bit, stop bit, baud rate, etc.
  - See LDD3 for details
- tiocmget and tiocmset
  - Control line setting set and get
  - See LDD3 for details

5. ioctls

- Some 70 different ioctls to ttys
- See summary in LDD3

6. proc and sysfs handling of TTY devices

- Automatic creation of /proc entries on definition of read_proc and write_proc callbacks.
- See LDD3

7. Core struct details

- See LDD3 for full details of the following structs:
  - struct tty_driver
  - struct tty_operations
  - struct tty_struct

Appendix A. Debugging drivers

1. Debugging support in the kernel
2. Manual techniques
3. Debugging tools
4. Performance measurement
5. Hardware tools

1. Debugging support in the kernel

- Built-in debugging capabilities
- "kernel hacking" menu
- CONFIG_DEBUG_KERNEL:
  - Required for enabling any other debug option
- CONFIG_DEBUG_SLAB:
  - Added checks in memory allocation functions
  - Useful for debugging memory overruns and missing initialization
  - Allocated buffers set to 0xa5
  - Freed buffers set to 0x6b

- Guard values put before and after allocated area: if modified => error
- CONFIG_DEBUG_PAGEALLOC:
  - Full pages removed from kernel when freed
  - CPU hog
  - May help in pinpointing memory corruption errors
- CONFIG_DEBUG_SPINLOCK:
  - Kernel catches errors on misuse of spinlocks
- CONFIG_DEBUG_SPINLOCK_SLEEP:
  - Checks for situations where sleeps are attempted while holding spinlocks

- CONFIG_INIT_DEBUG:
  - Checks for code attempting to access other code marked as __init and discarded after boot.
- CONFIG_DEBUG_INFO:
  - Generates kernel with full debug info (gcc -g).
  - Should configure CONFIG_FRAME_POINTER if using gdb
- CONFIG_MAGIC_SYSRQ:
  - Enables use of magic keyboard sequence to execute hardcoded commands in case of system hang.

- CONFIG_DEBUG_STACKOVERFLOW / CONFIG_DEBUG_STACK_USAGE:
  - Track down kernel stack overflows.
- CONFIG_KALLSYMS (in "General setup / Standard features):
  - Include kernel symbol table into kernel image.
  - Otherwise oopses are in hex.
- CONFIG_IKCONFIG / CONFIG_IKCONFIG_PROC (in "General setup"):

  - Include kernel configuration into kernel image and make it available through /proc.
- CONFIG_APIC_DEBUG (in "Power management / ACPI"):
  - Verbose ACPI debug info.
- CONFIG_DEBUG_DRIVER:
  - Turn on debug info in driver core.
- CONFIG_SCSCI_CONSTANTS (in "Device drivers / SCSI device support"):
  - Verbose SCSI debug info.

- CONFIG_INPUT_EVBUG:
  - Verbose logging for input devices (including keylogging).
- CONFIG_PROFILING (in "Profiling support"):
  - System performance tunning

2. Manual techniques

- printk

```
printk("Detected error 0x%x on interface %d:%d\n",
       error_code, iface_bus, iface_id);
```

- /proc
  - Main functions: include/linux/proc_fs.h

```
struct proc_dir_entry
    *create_proc_read_entry(const char *name,
                            mode_t mode,
                            struct proc_dir_entry *base,
                            read_proc_t *read_proc,
                            void * data)
void remove_proc_entry(const char *name,
                       struct proc_dir_entry *parent)
```

  - *read_proc* is a callback:

```
typedef int (read_proc_t)(char *page, char **start,
                          off_t off, int count,
                          int *eof, void *data);
```

- Write data to *page*
- Careful: can only fill 1 page at a time (4K)
- Use **start* to tell OS that you have more than a page. Your function will then be called more than once with a different *offset*.
- ALWAYS return the size you wrote. If you don't, nothing will be displayed.
- You can add your own /proc tree if you want ...

- ioctl:
  - Method implemented in most device driver models (char, block, net, fb, etc.) allowing custom functionality to be coded in driver ...

- Can extend your driver's ioctl to allow a user-space application to poll or change the driver's state outside of the OS' control.

- oops messages:
  - Report printed out by kernel regarding internal error that can't be handled.
  - Sometimes last output before system freeze => must be copied by hand.
  - Contains addresses and references to function addresses which can be understood by looking at System.map
  - Can be automatically decoded with klogd/ksymoops

```
Unable to handle kernel paging request at virtual address 0007007a
  printing eip:
 c022a8f6
 *pde = 00000000
 Oops: 0000
 CPU:    0
 EIP:    0010:[<c022a8f6>]    Not tainted
 EFLAGS: 00010202
 eax: 0000000a   ebx: 00000004   ecx: 00000001   edx: e3a74b80
 esi: 0007007a   edi: e62150fc   ebp: 0007007a   esp: dfbdbc8c
 ds: 0018   es: 0018   ss: 0018
 Process ip (pid: 2128, stackpage=dfbdb000)
 Stack: 00000000 bfff0018 00000018 0000000a 00000000 e41e8c00 c02f5acd e3a74b80
        c022aec1 e3a74b80 0000000a 00000004 0007007a c02f5ac8 00000e94 00000246
        e62150b4 00000000 e41e8c00 00000001 00000000 e5365a00 c022b039 e3a74b80
 Call Trace:    [<c022aec1>] [<c022b039>] [<c022df21>] [<c022e1d0>] [<c022b6ea>]
   [<c022afb0>] [<c022b1d0>] [<c0176f7e>] [<c022b2a0>] [<c022ddba>] [<c022d623>]
   [<c022db41>] [<c021c8f5>] [<c021daf3>] [<c0118238>] [<c021d44d>] [<c021e459>]
   [<c0107800>] [<c010770f>]

 Code: f3 a5 f6 c3 02 74 02 66 a5 f6 c3 01 74 01 a4 8b 5c 24 10 8b
```

- For good measure, always save the content of /proc/ksyms before generated an oops:

```
$ cat /proc/kallsyms > /tmp/kallsyms-dump
$ sync
```

- User-mode Linux: http://user-mode-linux.sf.net/
  - A version of Linux that runs entirely as a user-space process.
  - Ideal for prototyping and debugging new kernel functionality.
  - Widely used by kernel developers
  - Available starting in 2.5
  - UML is built as a new linux architecture (disable reiserfs)

```
$ cd ${PRJROOT}/kernel/uml/linux-2.6.11
$ make ARCH=um menuconfig
$ make linux ARCH=um
```

- Running user-mode Linux

```
$ ./linux
Checking for /proc/mm...not found
tracing thread pid = 14932
Linux version 2.6.11 (karim@localhost.localdomain) ...
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/ubd0
Calibrating delay loop... 2617.44 BogoMIPS
Memory: 29480k available
...
Initializing software serial port version 1
mconsole (version 2) initialized on /home/karim/.uml/...
unable to open root_fs for validation
UML Audio Relay (host dsp = /dev/sound/dsp, host mixer ...
Initializing stdio console driver
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 4096)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Cannot open root device "ubd0" or 62:00
Please append a correct "root=" boot option
Kernel panic: VFS: Unable to mount root fs on 62:00
```

3. Debugging tools

- gdb: all archs
  - Can use standard gdb to visualize kernel variables:

    ```
    $ gdb ./vmlinux /proc/kcore
    ```

  - Get more information when using “-g” flag
  - gdb grabs kcore snapshot at startup /no dynamic update
- kdb: x86 / ia64
  - Available from http://oss.sgi.com/
  - Integrated kernel debugger patch
  - Requires kernel patch (Linus allergic to kdb & friends)
  - Is used directly on the host being debugged
  - Access through PAUSE/BREAK key on keyboard

- IKD: Integrated Kernel Debugger
  - Available from ftp://ftp.kernel.org/pub/linux/kernel/people/andrea/ikd
  - Patch provides: stack-size watch / lock watch / ktrace
- kgdb: Full serial-line-based kernel debugger
  - Available from http://kgdb.sourceforge.net/
  - Sometimes available in kernel sources:

    arch/ppc/kernel/ppc-stub.c
  - Connect to remote target through gdb on host
  - Use gdb as you would for any other remote program

- LKCD: Linux Kernel Crash Dump
  - Available from http://lkcd.sourceforge.net/
  - Saves memory image at crash
  - Retrieve memory image at startup
  - Analyze image to find problems
- Kexec
- DProbes: Dynamic Probes
- Kprobes
- SystemTap

4. Performance measurement

- LMbench:
  - Available from http://www.bitmover.com/lmbench/
  - Runs heavy user-space benchmarks to determine kernel response time.
  - Not adapted for embedded (Requires Perl and C compiler.)
- kernprof:
  - Available from http://oss.sgi.com/projects/kernprof/
  - Kernel patch adding profiling data collection mechanism.
  - For x86 / ia64 / sparc64 / mips64

- Integrated sample-based profiler:
  - Activated upon passing “profile=” boot param
  - Profile data available in /proc/profile
  - User-space tools available from http://sourceforge.net/projects/minilop/
  - See BELS for how to install and use
- Measuring interrupt latency:
  - Two ways:
    - Self-contained
    - Induced

- Self-contained:
  - System's output connected to system input
  - Write driver with 2 main functions
  - Fire-up function:
    - Records current time
    - Trigger interrupt (write to output pin)
  - Interrupt handling function:
    - Records current time
    - Toggles output pin
  - Interrupt latency is measured using time difference
  - Can connect scope and observe timing
  - No need for time recording if using scope
  - Dead angle: can't see the time it takes to get to fire-up fct

- Induced:
  - Closer to reality
  - Interrupt generated by outside source (frequency generator)
  - Driver toggles output pin
  - Measure time between wave starts to get real latency
- Linux is not an RTOS ...
  - Try `ls -R /`

5. Hardware tools

- Oscilloscope
- Logic analyzer
- In-Circuit Emulator
- BDM/JTAG
  - Abatron / uses plain gdb
  - Wiggler / requires modified gdb
  - BDM4GDB (MPC860)
    http://bdm4gdb.sourceforge.net/
  - JTAG tool
    http://openwince.sourceforge.net/jtag/

Appendix B. Kernel data types

1. Use of standard C types
2. Assigning an explicit size to data types
3. Interface-specific types
4. Other portability issues
5. Linked lists

1. Use of standard C types

- Actual sizes differ between architectures
- See LDD3 p.289 for size of each type on various archs
- Pointers in kernel are "unsigned long" because pointers are the same size as that type on all supported archs.

2. Assigning an explicit size to data types

- <linux/types.h>
- Unsigned types:
  - u8
  - u16
  - u32
  - u64
- Signed types (rarely used):
  - s8
  - s16
  - s32

    - s64
- For headers exported to user-space, use these instead (no POSIX namespace pollution:
    - __u8, __s8
    - __u16, __s26
    - __u32, __s32
    - __u64, __s64
- If you want to be C99-compliant (and the compiler supports it):
    - uint8_t, uint16_t, uint32_t, uint64_t

3. Interface-specific types

- Commonly used data types are usually typedef'ed in the kernel
- Recently, typedefing has lost its appeal with kernel developers (opaque types)
- Many "_t" types defined in <linux/types.h> (size_t, pid_t, etc.)
- No problem when used in code: highly portable
- Problem when printing values out for debugging (usually such values need not be printed.)
- To print, cast to large possible type for typedefed "_t"

4. Other portability issues

- Avoid explicit constants: use #defines instead
- Time intervals:
  - jiffies not always 1000
  - Use HZ instead
- Page size:
  - PAGE_SIZE not always 4KB
  - Use PAGE_SHIFT to get number of bits to shift to get page numebr
  - <asm/page.h>

- Byte order:
  - PC is little-endian, but many platforms are big-endian
  - For internal driver use, no need to care
  - For outside communication:
    - <asm/byteorder.h>
    - Either __BIG_ENDIAN or __LITTLE_ENDIAN => use #ifdef
    - u32 cpu_to_le32(u32);
    - u32 le32_to_cpu(u32);
    - u32 cpu_to_be32(u32);
    - u32 be32_to_cpu(u32);

    - etc.
    - Varients with appended "s" for signed or "p" for pointer
- Data alignment:
    - On PC alignment is not a problem
    - For many non-PC architectures, non-aligned access is a problem
    - <asm/unaligned.h>
    - get_unaligned(ptr);
    - put_unaligned(val,ptr);
    - Type alignment in structures differs between archs

- Example alignment difference between processors in LDD3 p.294
- Structures padding may be inserted by compiler for performance, may cause problems when sharing structs with other hardware or systems.
- Use structure "packing" when necessary:

```
struct {} __attribute__ ((packed)) ...;
```

- Pointers and error values:
  - retval not always NULL on failure
  - Returning an error as a pointer value:

```
void *ERR_PTR(long error);
```

- Determining if pointer returned is error:

  long IS_ERR(const void *ptr);

- Retrieving error from ptr (after IS_ERR()):

  long PTR_ERR(const void *ptr);

5. Linked lists

- Kernel provides standard way to manage and maintain linked lists
- Should use kernel's primitives in as much as possible
- Primitives do not implement any locking, must use appropriate locking
- <linux/list.h>

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

- Include "struct list_head" as part of custom structs:

```
struct my_struct {
        struct list_head list;
        /* my stuff ... */
}
```

- Static initialization:
  - LIST_HEAD(my_list);
- Dynamic initialization:
  - struct list_head my_list;
  - INIT_LIST_HEAD(&my_list);

- list_add(struct list_head *new, struct list_head *head);
  - Add entry to list right after head
  - Could pass list entry instead of real head
- list_add_tail(struct list_head *new, struct list_head *head);
  - Add to end of list
- list_del(struct list_head *entry);
  - Remove from list

- list_del_init(struct list_head *entry);
  - Remove from list and reinit pointers
  - For removing and inserting in other lists
- list_move(struct list_head *entry, struct list_head *head);
  - Move entry to begining
- list_move_tail(struct list_head *entry, strust list_head *head);
  - Move entry to end

- list_empty(struct list_head *head);
  - retval is nonzero if empty
- list_splice(struct list_head *list, struct list_head *head);
  - Insert a list in other list
- list_entry(struct list_head *ptr, type_of_struct, field_name);
  - Use pointer arithmetic to get the pointer to the struct containing the list_head entry.
  - "type_of_struct" is the structure containing the "struct list_head". For example, my_struct.

- "field_name" is the name of "struct list_head" within custom struct.
- For example:

```
struct my_struct *my_ptr =
list_entry(list_ptr, struct my_struct, list);
```

- Macros for traversing lists:
  - list_for_each(struct list_head *cursor, struct list_head *list)
    - for() loop executed once for each list entry
    - Do not modify list in loop
  - list_for_each_prev(struct list_head *cursor, struct list_head *list)
    - Same as list_for_each() but in reverse

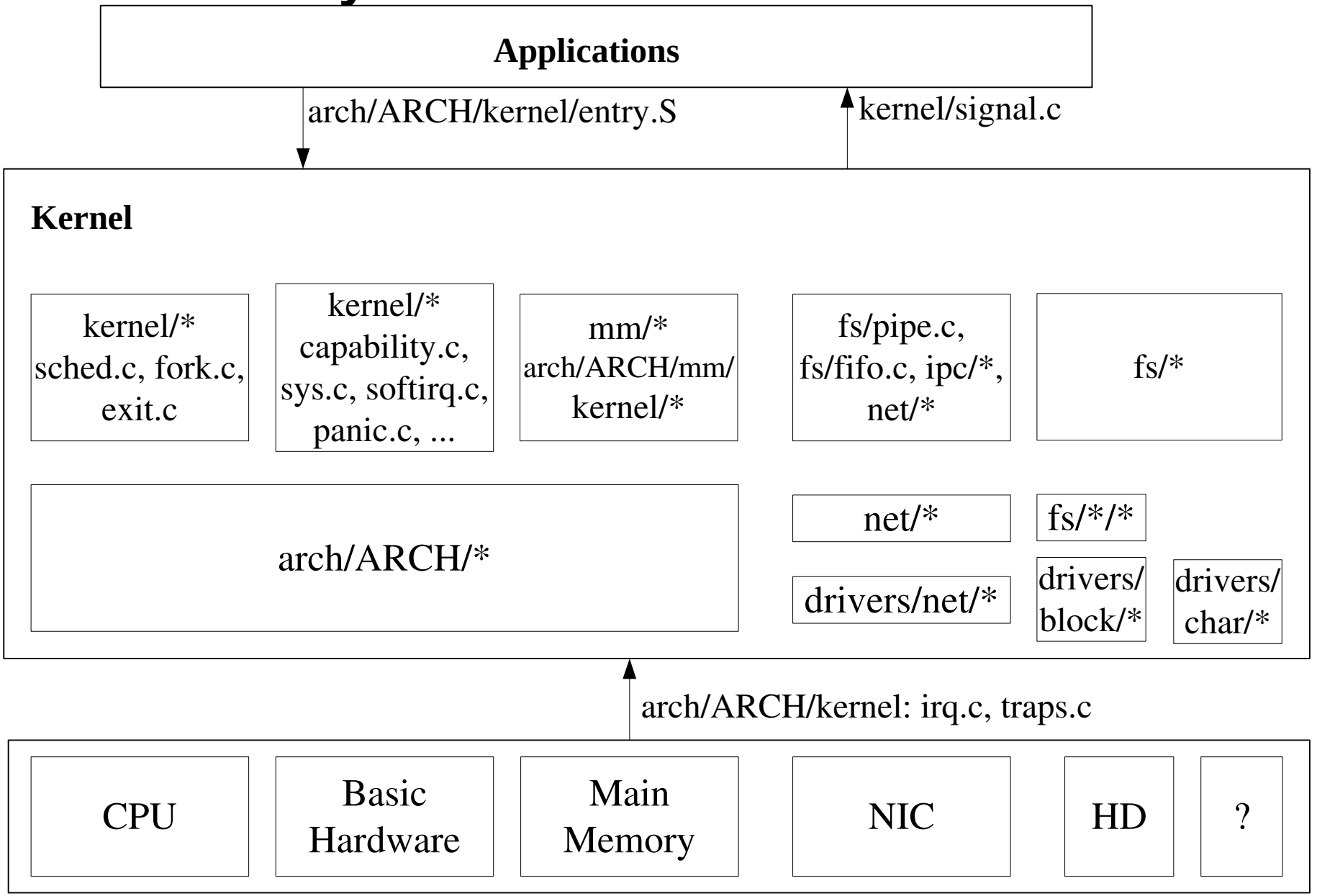
- list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)
  - Same as list_for_each() but saves next entry in list in case current entry is removed.
- list_for_each_entry(type *cursor, struct list_head *list, member)
- list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)
  - Same as before, but avoids having to use list_entry() in loop by implementing the functionality directly.
  - "cursor" is the pointer to the custom struct type
  - "member" is name of list in custom struct type

- Other type of list defined "hlist" => same, but head has only got one pointer.

Appendix C. Kernel integration

1. Kernel layout: Where are the drivers?
2. Kernel build system
3. Kernel config system
4. Adding a driver to the kernel sources
5. Creating patches
6. Distributing work and interfacing with the community

1. Kernel layout: Where are the drivers?



arch	45MB	=>	architecture-dependent functionality
Documentation	8MB	=>	main kernel documentation
drivers	**100MB**	**=>**	**all drivers**
fs	19MB	=>	virtual filesystem and all fs types
include	32MB	=>	complete kernel headers
init	108KB	=>	kernel startup code
ipc	172KB	=>	System V IPC
kernel	1.0MB	=>	core kernel code
mm	816KB	=>	memory management
net	10MB	=>	networking core and protocols
scripts	1.1MB	=>	scripts used to build kernel

- drivers/

```
acorn     cdrom     fc4         input
md        net       pcmcia      tc
acpi      char      firmware    isdn
media     nubus     pnp         telephony
atm       cpufreq   i2c         message
oprofile  s390      usb         base
crypto    ide       macintosh   misc
parisc    sbus      video       block
dio       ieee1394  mmc         parport
scsi      w1        bluetooth   eisa
pci       mca       mtd         infiniband
serial    zorro
```

2. Kernel build system

drivers/Makefile

```
#
# Makefile for the Linux kernel device drivers.
#
# 15 Sep 2000, Christoph Hellwig <hch@infradead.org>
# Rewritten to use lists instead of if-statements.
#

obj-$(CONFIG_PCI)      += pci/
obj-$(CONFIG_PARISC)   += parisc/
obj-y                  += video/
obj-$(CONFIG_ACPI_BOOT)+= acpi/
# PnP must come after ACPI since it will eventually need to check if acpi
# was used and do nothing if so
obj-$(CONFIG_PNP)      += pnp/

# char/ comes before serial/ etc so that the VT console is the boot-time
# default.
obj-y                  += char/

# i810fb and intelfb depend on char/agp/
obj-$(CONFIG_FB_I810)  += video/i810/
obj-$(CONFIG_FB_INTEL) += video/intelfb/

# we also need input/serio early so serio bus is initialized by the time
# serial drivers start registering their serio ports
obj-$(CONFIG_SERIO)    += input/serio/
obj-y                  += serial/
obj-$(CONFIG_PARPORT)  += parport/
obj-y                  += base/ block/ misc/ net/ media/
obj-$(CONFIG_NUBUS)    += nubus/
obj-$(CONFIG_ATM)      += atm/
obj-$(CONFIG_PPC_PMAC) += macintosh/
obj-$(CONFIG_IDE)      += ide/
```

drivers/char/Makefile

```
#
# Makefile for the kernel character device drivers.
#

#
# This file contains the font map for the default (hardware) font
#
FONTMAPFILE = cp437.uni

obj-y    += mem.o random.o tty_io.o n_tty.o tty_ioctl.o

obj-$(CONFIG_LEGACY_PTYS)   += pty.o
obj-$(CONFIG_UNIX98_PTYS)   += pty.o
obj-y                       += misc.o
obj-$(CONFIG_VT)            += vt_ioctl.o vc_screen.o consolemap.o \
   consolemap_deftbl.o selection.o keyboard.o
obj-$(CONFIG_HW_CONSOLE)    += vt.o defkeymap.o
obj-$(CONFIG_MAGIC_SYSRQ)   += sysrq.o
obj-$(CONFIG_ESPSERIAL)     += esp.o
obj-$(CONFIG_MVME147_SCC)   += generic_serial.o vme_scc.o
obj-$(CONFIG_MVME162_SCC)   += generic_serial.o vme_scc.o
obj-$(CONFIG_BVME6000_SCC)  += generic_serial.o vme_scc.o
obj-$(CONFIG_ROCKETPORT)    += rocket.o
obj-$(CONFIG_SERIAL167)     += serial167.o
obj-$(CONFIG_CYCLADES)      += cyclades.o
obj-$(CONFIG_STALLION)      += stallion.o
obj-$(CONFIG_ISTALLION)     += istallion.o
obj-$(CONFIG_DIGIEPCA)      += epca.o
obj-$(CONFIG_SPECIALIX)     += specialix.o
obj-$(CONFIG_MOXA_INTELLIO) += moxa.o
obj-$(CONFIG_A2232)         += ser_a2232.o generic_serial.o
obj-$(CONFIG_ATARI_DSP56K)  += dsp56k.o

...
```

- For full details:
  - Documentation/kbuild/makefiles.txt

3. Kernel config system

drivers/Kconfig

```
# drivers/Kconfig

menu "Device Drivers"

source "drivers/base/Kconfig"

source "drivers/mtd/Kconfig"

source "drivers/parport/Kconfig"

source "drivers/pnp/Kconfig"

source "drivers/block/Kconfig"

source "drivers/ide/Kconfig"

source "drivers/scsi/Kconfig"

source "drivers/cdrom/Kconfig"

source "drivers/md/Kconfig"

source "drivers/message/fusion/Kconfig"

source "drivers/ieee1394/Kconfig"

source "drivers/message/i2o/Kconfig"

source "drivers/macintosh/Kconfig"

source "net/Kconfig"

source "drivers/isdn/Kconfig"
```

drivers/char/Kconfig

```
#
# Character device configuration
#

menu "Character devices"

config VT
bool "Virtual terminal" if EMBEDDED
select INPUT
default y if !VIOCONS
---help---
  If you say Y here, you will get support for terminal devices with
      ...

config VT_CONSOLE
bool "Support for console on virtual terminal" if EMBEDDED
depends on VT
default y
---help---
  The system console is the device which receives all kernel messages
      ...

config HW_CONSOLE
bool
depends on VT && !S390 && !UML
default y

config SERIAL_NONSTANDARD
bool "Non-standard serial port support"
---help---
  Say Y here if you have any non-standard serial boards -- boards
      ...
```

- For full details:
  - Documentation/kbuild/kconfig-language.txt

4. Adding a driver to the kernel sources

- Create directory in drivers/ or drivers/*
- Put driver sources in created directory
- Modify drivers/Makefile or drivers/*/Makefile to recognize new directory.
- Add proper Makefile to your driver's directory
- Modify drivers/Kconfig or drivers/*/Kconfig to show driver in kernel config.
- Add proper Kconfig to your driver's directory

5. Creating patches

- Patch basics
  - A patch is a file containing differences between a certain “official” kernel version and a modified version.
  - Patches are most commonly created using a command line that looks like:

```
$ diff -urN original-kernel modified-kernel > my_patch
```

  - Patches can be incremental (e.g. need to apply patches A, B and C before applying patch D)
  - Patches will easily break if not applied to the exact kernel version they were created for.

- Analyzing a patch's content:

```
$ diffstat -p1 my_patch
```

- Testing a patch before applying it:

```
$ cp my_patch ${PRJROOT}/kernel/linux-2.6.11
$ cd ${PRJROOT}/kernel/linux-2.6.11
$ patch --dry-run -p1 < my_patch
```

- Applying patches:

```
$ patch -p1 < my_patch
```

6. Distributing work and interfacing with the community

- Create project webpage (possibly on sourceforge)
- Post patches to LKML
  - Title:
    - [PATCH] n/x
    - [PATCH/RFC] n/x
  - “signed-off-by: Your Name <your@mail.tld>”
- Integrate community feedback
- Continue posting updated patches

Appendix D. Porting the kernel

1. Per-arch kernel layout
2. Kernel startup
3. Key definitions
4. Interfacing between bootloader and OS

1. Per-arch kernel layout

```
$ ll arch/ppc
total 104
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 4xx_io
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 8260_io
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 8xx_io
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 amiga
drwxr-xr-x   10 karim    karim        4096 Jun 17 15:48 boot
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 configs
-rw-r--r--    1 karim    karim       36331 Jun 17 15:48 Kconfig
-rw-r--r--    1 karim    karim        1747 Jun 17 15:48 Kconfig.debug
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 kernel
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 lib
-rw-r--r--    1 karim    karim        4475 Jun 17 15:48 Makefile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 math-emu
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mm
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 oprofile
drwxr-xr-x    5 karim    karim        4096 Jun 17 15:48 platforms
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 syslib
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 xmon
```

```
$ ll arch/mips/
total 220
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 arc
drwxr-xr-x   12 karim    karim        4096 Jun 17 15:48 au1000
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 boot
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 cobalt
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 configs
drwxr-xr-x    6 karim    karim        4096 Jun 17 15:48 ddb5xxx
drwxr-xr-x    4 karim    karim        4096 Jun 17 15:48 dec
-rw-r--r--    1 karim    karim       18490 Jun 17 15:48 defconfig
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 galileo-boards
drwxr-xr-x    5 karim    karim        4096 Jun 17 15:48 gt64120
drwxr-xr-x    5 karim    karim        4096 Jun 17 15:48 ite-boards
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 jazz
drwxr-xr-x    4 karim    karim        4096 Jun 17 15:48 jmr3927
-rw-r--r--    1 karim    karim       42476 Jun 17 15:48 Kconfig
-rw-r--r--    1 karim    karim        2564 Jun 17 15:48 Kconfig.debug
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 kernel
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 lasat
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 lib
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 lib-32
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 lib-64
-rw-r--r--    1 karim    karim       21398 Jun 17 15:48 Makefile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 math-emu
drwxr-xr-x    6 karim    karim        4096 Jun 17 15:48 mips-boards
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mm
drwxr-xr-x    6 karim    karim        4096 Jun 17 15:48 momentum
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 oprofile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 pci
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 pmc-sierra
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 sgi-ip22
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 sgi-ip27
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 sgi-ip32
drwxr-xr-x    5 karim    karim        4096 Jun 17 15:48 sibyte
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 sni
drwxr-xr-x    4 karim    karim        4096 Jun 17 15:48 tx4927
```

```
$ ll arch/arm/
total 156
drwxr-xr-x    4 karim    karim        4096 Jun 17 15:48 boot
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 common
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 configs
-rw-r--r--    1 karim    karim       21624 Jun 17 15:48 Kconfig
-rw-r--r--    1 karim    karim        3847 Jun 17 15:48 Kconfig.debug
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 kernel
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 lib
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-clps711x
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-clps7500
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ebsa110
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-epxa10db
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-footbridge
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-h720x
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-imx
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-integrator
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-iop3xx
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ixp2000
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ixp4xx
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-l7200
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-lh7a40x
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-omap
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-pxa
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-rpc
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-s3c2410
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-sa1100
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-shark
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-versatile
-rw-r--r--    1 karim    karim        7636 Jun 17 15:48 Makefile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mm
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 nwfpe
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 oprofile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 tools
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 vfp
```

```
$ ll arch/i386/
total 140
drwxr-xr-x    4 karim    karim        4096 Jun 17 15:48 boot
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 crypto
-rw-r--r--    1 karim    karim       26742 Jun 17 15:48 defconfig
-rw-r--r--    1 karim    karim       42227 Jun 17 15:48 Kconfig
-rw-r--r--    1 karim    karim        2255 Jun 17 15:48 Kconfig.debug
drwxr-xr-x    5 karim    karim        4096 Jun 17 15:48 kernel
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 lib
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-default
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-es7000
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-generic
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-visws
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-voyager
-rw-r--r--    1 karim    karim        6341 Jun 17 15:48 Makefile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 math-emu
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mm
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 oprofile
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 pci
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 power
```

2. Kernel startup

Explanation for TQM860 PPC board

0.Kernel entry point:

arch/ppc/boot/common/crt0.S:*_start*

1. *_start* calls on:

arch/ppc/boot/simple/head.S:*start*

2. *start* calls on:

arch/ppc/boot/simple/relocate.S:*relocate*

3. *relocate* calls on:

arch/ppc/boot/simple/misc-embedded.c: *load_kernel()*

4. *load_kernel()* initializes the serial line and uncompresses kernel starting at address 0.

6. *load_kernel()* returns to *relocate*
7. *relocate* jumps to address 0x00000000, where kernel start address is.
8. arch/ppc/kernel/head_8xx.S: *__start*
9. *__start* eventually calls init/main.c:*start_kernel()*
10. *start_kernel()* does:
    1. Locks kernel
    2. *setup_arch()*
    3. *sched_init()*
    4. *parse_args()*
    5. *trap_init()*
    6. *init_IRQ()*

7. *time_init()*
8. *console_init()*
9. *mem_init()*
10. *calibrate_delay()* => *loops_per_jiffy*
11. *rest_init()*

11. *rest_init()* does:
    1. Start `init` thread
    2. Unlocks the kernel
    3. Becomes the idle task

12. The `init` task:
    1. *lock_kernel()*
    2. *do_basic_setup()* => call various *init()* fcts
    3. *prepare_namespace()* => mount rootfs
    4. *free_initmem()*
    5. *unlock_kernel()*
    6. *execve()* on the init program (`/sbin/init`)

3. Key definitions

```
$ ll include/
total 148
   ...
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-alpha
drwxr-xr-x   24 karim    karim        4096 Jun 17 15:48 asm-arm
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-arm26
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 asm-cris
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-frv
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-generic
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-h8300
drwxr-xr-x   10 karim    karim        4096 Jun 17 15:48 asm-i386
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 asm-ia64
drwxr-xr-x    5 karim    karim        4096 Jun 17 15:48 asm-m32r
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-m68k
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-m68knommu
drwxr-xr-x   45 karim    karim        4096 Jun 17 15:48 asm-mips
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-parisc
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-ppc
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 asm-ppc64
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-s390
drwxr-xr-x   31 karim    karim        4096 Jun 17 15:48 asm-sh
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-sh64
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-sparc
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-sparc64
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-um
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-v850
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 asm-x86_64
drwxr-xr-x   18 karim    karim       12288 Jun 17 15:48 linux
   ...
```

```
$ ll include/asm-mips/
   ...
-rw-r--r--    1 karim    karim         519 Jun 17 15:48 m48t35.h
-rw-r--r--    1 karim    karim         696 Jun 17 15:48 m48t37.h
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-atlas
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-au1x00
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-db1x00
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ddb5074
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-dec
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ev64120
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ev96100
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-generic
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ip22
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ip27
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ip32
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ja
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-jazz
drwxr-xr-x    3 karim    karim        4096 Jun 17 15:48 mach-jmr3927
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-lasat
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-mips
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ocelot
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-ocelot3
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-pb1x00
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-rm200
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-sibyte
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-vr41xx
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mach-yosemite
-rw-r--r--    1 karim    karim        1608 Jun 17 15:48 marvell.h
-rw-r--r--    1 karim    karim         450 Jun 17 15:48 mc146818rtc.h
-rw-r--r--    1 karim    karim        4189 Jun 17 15:48 mc146818-time.h
drwxr-xr-x    2 karim    karim        4096 Jun 17 15:48 mips-boards
   ...
```

```
$ ll include/asm-mips/vr41xx/
total 52
-rw-r--r--    1 karim    karim        1489 Jun 17 15:48 capcella.h
-rw-r--r--    1 karim    karim        1856 Jun 17 15:48 cmbvr4133.h
-rw-r--r--    1 karim    karim        1497 Jun 17 15:48 e55.h
-rw-r--r--    1 karim    karim        1174 Jun 17 15:48 mpc30x.h
-rw-r--r--    1 karim    karim        2559 Jun 17 15:48 pci.h
-rw-r--r--    1 karim    karim        1417 Jun 17 15:48 siu.h
-rw-r--r--    1 karim    karim        1411 Jun 17 15:48 tb0219.h
-rw-r--r--    1 karim    karim        1439 Jun 17 15:48 tb0226.h
-rw-r--r--    1 karim    karim        6151 Jun 17 15:48 vr41xx.h
-rw-r--r--    1 karim    karim        6728 Jun 17 15:48 vrc4173.h
-rw-r--r--    1 karim    karim        1492 Jun 17 15:48 workpad.h
```

4. Interfacing between bootloader and OS

- Very bootloader dependent
- Information sources:
  - U-Boot documentation has good explanation of interface between it and Linux.
  - Comments and code in Linux sources for your arch