Linux Device Drivers

Björne Lindberg 04/02/06

Linux Device Driver

Device	Hardware, real or abstract
Driver	software

Compiled into kernel / Dynamic loading at runtime ➔ modules

Modules

Object code .ko

Loads with insmod, modprobe ;modprobe checks for dependencies and loads also dependent modules
Unloads with rmmod
Show with lsmod

Classes of drivers

Char sequential access
Block random access
Network

Version numbering

Rules!

No floating point
User space-libraries not reachable from kernel
No delays!!
No protection from overwriting other memory areas!!

Example tutorial.c:

```
#include <linux/device.h>
MODULE_LICENSE("GPL");

static int tutorial_init(void)  {
        printk(KERN_ALERT "Init module\n");
        return 0;
}
```

```
static void tutorial_exit(void)  {
        printk(KERN_ALERT "Exit module\n");
}

module_init(tutorial_init);
module_exit(tutorial_exit);
```

Makefile:

```
ifneq ($(KERNELRELEASE),)
# call from kernel build system
obj-m   := tutorial.o

else

KERNELDIR ?= /lib/modules/2.6.10/build
PWD       := $(shell pwd)

default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD)
LDDINCDIR=$(PWD)/../include modules
endif
```

Devices

To use a driver we need an “entry” in /dev/

mknod	/dev/my_nod	c	major	minor
Major nr	Identifies the driver			
Minor nr	Identifies the device (used by the driver)			

Allocating and freeing Device Numbers

```
dev_t dev;
```

```
MAJOR(dev);
MINOR(dev);
dev = MKDEV(int major, int minor);

int register_chrdev_region(dev_t first, unsigned int
        count, char *name);
int alloc_chrdev_region(dev_t *dev, unsigned int
        firstminor, unsigned int count, char *name);
```

Returns:

< 0;	error
0;	success and the driver has been assigned major & minor number dev

name
The name associated with the driver. /proc/modules

void unregister_chrdev_region(dev_t first, unsigned int count);

Important kernel data structures

- file_operations
- file
- inode
- cdev

file_operations

Connects system calls with our drivers code by providing pointers to the drivers functions.

```
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);
        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 __user *);
        ssize_t (*sendpage) (struct file *, struct
                page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file
                *, unsigned long, unsigned long,
                unsigned long, unsigned long);   };
```

Assigning function pointers to file_operations

```
struct file_operations tutorial_fops = {
        .owner  =       THIS_MODULE,
        .llseek =       tutorial_llseek,
        .read   =       tutorial_read,
        .write  =       tutorial_write,
        .ioctl  =       tutorial_ioctl,
        .open   =       tutorial_open,
        .release =      tutorial_release,
};
```

file

Represents *open* files in the kernel.

DON'T mix with disk files, represented by a struct inode
Created by the kernel on open call and are passed to any function, operating on the file.

```
struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount          *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        struct fownfile_struct  f_owner;
        unsigned int            f_uid, f_gid;
        int                     f_error;
        struct file_ra_state    f_ra;
        unsigned long           f_version;
        void                    *f_security;
        /* needed for tty driver, and maybe others */
        void                    *private_data;
        /* Used by fs/eventpoll.c to link all the
                hooks to this file */
        struct list_head        f_ep_links;
        spinlock_t              f_ep_lock;
};
```

From "Linux device driver 3rd edition

mode_t f_mode;

The file mode identifies the file as either readable or writable (or both), by means of the bits `FMODE_READ` and `FMODE_WRITE`. You might want to check this field for read/write permission in your *ioctl* function, but you don't need to check permissions for *read* and *write* because the kernel checks before invoking your method. An attempt to write without permission, for example, is rejected without the driver even knowing about it.

loff_t f_pos;

The current reading or writing position. `loff_t` is a 64-bit value (`long long` in *gcc* terminology). The driver can read this value if it needs to know the current position in the file, but should never change it (*read* and *write* should update a position using the pointer they receive as the last argument instead of acting on `filp->f_pos` directly).

unsigned int f_flags;

These are the file flags, such as `O_RDONLY`, `O_NONBLOCK`, and `O_SYNC`. A driver needs to check the flag for nonblocking operation, while the other flags are seldom used. In particular, read/write permission should be checked using `f_mode` instead of `f_flags`. All the flags are defined in the header `<linux/fcntl.h>`.

`struct file_operations *f_op;`

The operations associated with the file. The kernel assigns the pointer as part of its implementation of *open*, and then reads it when it needs to dispatch any operations. The value in `filp->f_op` is never saved for later reference; this means that you can change the file operations associated with your file whenever you want, and the new methods will be effective immediately after you return to the caller. For example, the code for *open* associated with major number 1 (*/dev/null*, */dev/zero*, and so on) substitutes the operations in `filp->f_op` depending on the minor number being opened. This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of "method overriding" in object-oriented programming.

`void *private_data;`

The *open* system call sets this pointer to `NULL` before calling the *open*method for the driver. The driver is free to make its own use of the field or to ignore it. The driver can use the field to point to allocated data, but then must free memory in the *release* method before the `file` structure is destroyed by the kernel. `private_data` is a useful resource for preserving state information across system calls and is used by most of our sample modules.

`struct dentry *f_dentry;`

The directory entry (*dentry*) structure associated with the file. Dentries are an optimization introduced in the 2.1 development series. Device driver writers normally need not concern themselves with dentry structures, other than to access the `inode` structure as `filp->f_dentry->d_inode`.

inode

Represents files.

Lots of fields but mostly 2 fields of interest for drivers if it is a device file:

- dev_t i_rdev;
- struct cdev *i_cdev; cdev's an internal kernel stucture, representing char devices

cdev

Represents the char devices

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        struct file_operations *ops;
        struct list_head list;
};
```

Registration of char devices and the operations on it

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```
Initiates the cdev structures

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```
Register the device.

```
void cdev_del(struct cdev *dev);
```

Using resources

- request_region(port, range, “My_region”);
- release_region(port, range);
- request_irq(…….);
- free_irq(…..);

Open

Tasks:

Check for device-specific errors
Initialise the device,
If different fops, update *fops, according to minor nr
Allocate (and fill) datastructures to be put in filp->private_data (ex semaphores etc)

```
int open(const char *path, int flags);

int my_open(struct inode *inode, struct file *filp)
{
        ......
        return 0;
}
```

Check for minor nr

```
MINOR(inode->i_rdev);
```

Release

Invoked when the file structure is being released (Called by close when necessary to match open).

Tasks:
Deallocate anything open has allocated
Shutdown the device on last close

Structure of a module.c-file

```
My_module_init()
{
        Allocate resources, initiate interrupts,
        register driver etc
}

my_module_exit()
{
        Deallocate anything, allocated by
        my_module_init
}

struct file_operations tutorial_fops = {
        .owner =    THIS_MODULE,
        .llseek =   tutorial_llseek,
        .read =     tutorial_read,
        .write =    tutorial_write,
        .ioctl =    tutorial_ioctl,
        .open =     tutorial_open,
        .release =  tutorial_release,
};

/*Implement all necessary system functions*/
my_open(…)
{
.
.
}
```

my_read(…)
{
.
.
}

module_init(my_module_init);
module_exit(my_module_exit);

tutorial_init

allocate resources: IO-ports Memory Interrupts

Ex:

```
int tutorial_init(void)
{
        int result; /* Get our needed resources. */
        result = check_region(BASEADRESS_PORT,
        NR_OF_PORTS);
        if (result) {
            printk(KERN_INFO "tutorial: can't get I/O
                port address 0x%lx\n", parport_base);
            return result;
        } request_region(BASEADRESS_PORT, NR_OF_PORTS,
                "parport");
        printk(KERN_ALERT "Init module\n");
        if (major) {
                dev=MKDEV(major, minor);
                result = register_chrdev_region(dev,
                        nr_devs, "tutorial_driver");
        } else {
                result = alloc_chrdev_region(&dev, minor,
                        nr_devs, "tutorial_driver");
        }
        if(result < 0) {
                printk(KERN_WARNING "Cant get major nr %
                        d\n", major);
                return result;
        } else {
                major = MAJOR(dev);
                minor = MINOR(dev);
                printk(KERN_ALERT "Got major nr %d and
                        minor nr %d\n", major, minor);
        }
        cdev_init(&tutorial_cdev, &tutorial_fops);
        cdev_add(&tutorial_cdev, dev, 1);
       result = request_irq(my_irq, my_sh_interrupt,
              SA_SHIRQ | SA_INTERRUPT,"my_module",
                tutorial_sh_interrupt);
       if (result) {
```

```
                printk(KERN_INFO "my_module: can't get
                assigned irq %i\n", my_irq);
                tutorial_irq = -1;
        } return 0;
}
```

tutorial_exit

Deallocate resources
IO-ports
Memory
Interrupts

```
void tutorial_exit(void) {
        if (my_irq >= 0) {
                outb(0x0, short_base + 2);   /*
                        disable the
                        interrupt */
                free_irq(my_irq, my_sh_interrupt);
        }
        release_region(BASEADRESS_PORT, NR_OF_PORTS);
        printk(KERN_ALERT "Exit module\n");
        unregister_chrdev_region(dev, nr_devs);
        cdev_del(&tutorial_cdev);



}
```

Concurrency & Race conditions

Problems:

- Shared resources
- Resources comes and goes

Implement atomic operations by using locks and define “critical sections”
Dont make any resources available before its in a state of proper function.
Track usage

Semaphores

Protect critical code/data, transfers etc

```
struct semaphore my_sem;
```

Value:
1 ==> available
0 ==> not available

```
void sema_init(struct semaphore *sem, int val);
```

val is initial value of the semaphore

```
ex:      void sema_init(&my_sem, 1);
```

Obtain a semaphore

```
void down(&my_sem);
```

If not available, the process will sleep until the semaphore is freed.
Dont use if possible!!

```
int down_interruptible(&my_sem);
```

If not available, the process will sleep until the semaphore is freed or interrupted
Returns 0 if/when success and semaphore available
Returns 1 if interrupted by a signal. NEEDS CHECK THE RETURN VALUE!

```
ex: if (down_interruptible(&my_sem))
        return -ERESTARTSYS;
```

```
int down_trylock(&my_sem);
```

Returns 0 if/when success and semaphore available

Release a semaphore

```
void up(&my_sem);
```

DO NOT MISS THIS!!

Transfer data between user and kernel space

Check if valid addresses

int access_ok(int type, const void *useraddr, unsigned long size);

Checks if useraddr is within the process virtual memory area

Copying

int get_user(void *to, const void *useraddr);
Copy sizeof(useraddr) bytes from user space to “to”. Checks if valid addresses.

int put_user(void *to, const char *kerneladdr);
Copy sizeof(kerneladdr) bytes from kernel space to “to”.

int __get_user(void *to, const void *useraddr);
Copy sizeof(useraddr) bytes from user space to “to”. Don’t checks if valid addresses.

int __put_user(void *to, const char *kerneladdr);
Copy sizeof(kerneladdr) bytes from kernel space to “to”. Don’t checks if valid addresses.

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

unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
Copy count amount of data to or from the specified pointers

return amount of data still to be copied:

0:	Success
>0:	The amount of data, succeccfully copied was less than count. Return amount of data, still to be copied

The data area, addressed in user space, can be swapped out from memory and the page-fault handler can put the process to sleep while the data is being transferred into place (ex. from swap space). These function might need to be protected with semaphores!!!

read

size_t read(int fd, void *buf, size_t count);

ssize_t my_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
}

Implement this behavior:

At success, the requested number of bytes has been transferred and the return value equals count.
If only part of the data has been transferred, the return value equals transferred number of bytes (return value < count)
If end-of-file was reached, return 0.
Return < 0 when error. (Ex –EINTR; interrupted system call or –EFAULT; bad address)
If NO data available in the input buffer, the driver must by default block (go asleep) until at least one byte is there. If O_NONBLOCK is set, return immediately (don't block) with return value –EAGAIN.

Ex:
```
ssize_t my_read( struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
        my_data_struct *data = filp->private_data;
        .
        .
if(down_interruptible(&data->sem)
                return –ERESTARTSYS;
        .
        .
        if(copy_to_user(buf, data->data, count)
        {
                up(&data->sem);
                return –EFAULT;
        }
        up(&data->sem);
        return count;
}
```

write

```
size_t write( int fildes, const void *buf, size_t count);

ssize_t my_write( struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
.
}
```

Implement this behavior:

At success, the requested number of bytes has been transferred and the return value equals count.
If only part of the data has been transferred, the return value equals transferred number of bytes (return value < count)
If nothing was written, return 0.
Return < 0 when error. (Ex –EINTR; interrupted system call or –EFAULT; bad address)
If the output buffer is full, the driver must by default block (go asleep) until some space is freed. If O_NONBLOCK is set, return immediately (don't block) with return value –EAGAIN.
If the device cannot accept any more data, return with –ENOSPC.

Allocating memory

kmalloc

void *kmalloc(size_t size, priority);
32 byte < memsize < 32 pages, 128 KB, physically contiguous.
Allocated in chunks with size multiple of 2.

priority:
GFP_KERNEL; Can sleep ➔ Be scheduled out; for
"normal operations"
GFP_ATOMIC; Can't sleep; Extra ordinary situations ex
in interrupt rutines

kfree(const void *addr)

get_free_page

returns pointers to new pages or the first byte of several pages

```
unsigned long get_zeroed_page(int priority);
unsigned long __get_free_page(int priority);
unsigned long __get_free_pages(int priority, unsigned int
order);
unsigned long get_dma_pages(int priority, unsigned long
order);
```

These pages are consecutive in physical memory

```
 unsigned long get_free_pages(int priority, unsigned long
order, int dma);
```

order: Powers of two of the number of pages wanted
priority: Same as kmalloc

void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);

vmalloc

```
void *vmalloc(unsigned long size);
```

Only accessed in kernel/CPU space, Virtual continuous (Beyond the physical memory)
Returns 0 if error.

```
vfree(void *addr)
```

Putting processes to sleep on a queue

Initialise queues:

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

alt

DECLARE_WAIT_QUEUE_HEAD(my_queue);

Functions:

don't use:

```
sleep_on(wait_queue_head_t *my_queue);
interruptible_sleep_on(wait_queue_head_t *my_queue);
sleep_on_timeout(wait_queue_head_t *my_queue,
                           long timeout);*in jiffies*/
interruptible_sleep_on_timeout(wait_queue_head_t *my_queue,
                           long timeout);*in jiffies*/
```

```
Use:
void wait_event(wait_queue_head_t *my_queue, int
                           condition);
int wait_event_interruptible(wait_queue_head_t *my_queue, int
                           condition);
int wait_event_timeout(wait_queue_head_t *my_queue, long
timeout);*in jiffies*/
int wait_event_interruptible_timeout(wait_queue_head_t
*my_queue, int condition, long timeout);*in jiffies*/
```

Waking up sleeping processes:

```
wake_up(wait_queue_head_t *my_queue);
wake_up_interruptible(wait_queue_head_t *my_queue);
wake_up_sync(wait_queue_head_t *my_queue);
wake_up_interruptible_sync(wait_queue_head_t
                                       *my_queue);
```

```
static int flag = 0;;


tutorial_read(..){
        wait_event_interruptible(my_queue, flag != 0);
        flag = 0;
        .
        .
}

tutorial_write(..){
        .
        flag = 1;
        wake_up_interruptible(&my_queue);
```

```
          .
          .
}
```

wait_event_interruptible() returns
0 if woken bay wake_up
> 0 if woken by interrupt

```
if(wait_event_interruptible(my_queue, flag != 0))
        return -ERESTARTSYS;
.
.
```

Working with I/O-ports & I/O-memory

Operations on I/O-port & I/O-memory have sideeffects
Operations on memory don't.
Look up for compiler optimization and caching of operations!!
Also available in user space!!

Use memory barriers:

void barrier();   void rmb();   void wmb();   void mb();

I/O-port & I/O-memory must be allocated before use

Allocate Ports

```
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start, unsigned
long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

Accessing ports

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

```
unsigned inl(unsigned port);
void outl(unsigned longword byte, unsigned port);

unsigned insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

Reads/writes count bytes starting at address addr. Data is read from or written to the single port port

```
unsigned insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);

unsigned insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

Allocate I/O-memory

```
int check_memory_region( unsigned long start, unsigned long
len);
void request_memory_region( unsigned long start, unsigned long
len, char *name);
void release_memory_region(unsigned long start, unsigned long
len);
```

Accessing I/O-memory

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);

void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

```
#define TUTORIAL_NR_PORTS       8
static unsigned long base = 0x378;

if (! request_region(base, TUTORIAL_NR_PORTS, "tutorial")) {
        printk(KERN_INFO "Can't get I/O port address 0x%lx\n",
                                base);
return -ENODEV;
}
```

unsigned long port = base;
unsigned char *ptr;

*ptr= inb(port);
rmb();

ioctl

Controlling the device

```
int ioctl(int fd, int cmd, …);
int ioctl(int fd, int cmd, void *arg);

int my_ioctl(struct inode *inode, struct file *filp,
               unsigned int cmd, unsigned long arg)
{
.
}
```

The dots don't represent a variable nr of argument!!
Represents one argument, mostly a char *arg or void *arg.

The command number, consists of a 16 bit number, should be unique across the system!!
Look in /your_kernelsourcepath/Documentation/ioctl-number.txt for conventions to choose cmd-number

Makros to construct cmd-number:

_IO(type, nr)
_IOR(type, nr, dataitem)
_IOW(type, nr, dataitem)
_IORW(type, nr, dataitem)

Return –ENOTTY if no matching cmd

OBS!!

- If arg is a value: No problems!!
  But pointers must be checked with access_ok(…) or equal!!
- Predefined commands overrides your function!!!!

Capabilities

Sets permissions for privileged operations we can use in drivers.
See <Linux/capabilities> for full set of capabilities

```
Int capable(int capability);
```
EX:
```
#DEFINE   MY_CMDR     _IOR('k', 1, mydata)
#DEFINE   MY_CMDW     _IOW('k', 2, mydata)
#DEFINE   MY_CMDSET   _IOW('k', 3)

ioctl(fd, MY_CMD, &mydata);

int my_ioctl(struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
        .
        switch(cmd) {
                case MY_CMDR:
                .
                .
                break;

                case MY_CMDW:
                .
        .
                break;

                case MY_CMDSET:
                if(!capable(CAP_SYS_ADMIN)
                        return -EPERM;
                .
                .
                break;

                default:  /* redundant, as cmd should
                        be checked against MAXNR */
                return -ENOTTY;
        }
```

```
        return ret;
}
```

Rules/Tips & Recommendations

Avoid floating point arithmetics; If you must use them, you must save FP state on the stack yourself!!
Avoid polling in drivers
Avoid delays!!
Don't try to be too clever!!
User-space functions cannot be reached from kernelspace (no libc)
Kernel code will not be interrupted by the scheduler!!

Don't invite to Buffer overrun!!

Interrupts

ISR Interrupt service Routine

Interrupt without ISR will be ignored

Installed IRQ handlers in /proc/interrupts

```
           CPU0
  0:      12675          XT-PIC  timer
  1:        205          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  3:         59          XT-PIC  eth0
  5:          2          XT-PIC  ehci-hcd, ohci1394
  8:          1          XT-PIC  rtc
  9:          0          XT-PIC  usb-uhci
 10:          0          XT-PIC  usb-uhci
 11:          0          XT-PIC  usb-uhci, usb-uhci
 12:         30          XT-PIC  PS/2 Mouse
 14:       3943          XT-PIC  ide0
 15:         44          XT-PIC  ide1
NMI:          0
ERR:          0
```

More statistics in /proc/stat

```
cpu  289 0 666 16088
cpu0 289 0 666 16088
page 27171 12777
swap 1 0
intr 22101 17043 587 0 79 3 2 4 0 1 0 0 0 30 0 4308 44
disk_io: (3,0):(568,554,1661,14,29) (3,1):(3715,2407,52058,1308,25504)
ctxt 22912
btime 1078209302
processes 4349
```

Allocate Interrupts

```
int request_irq(unsigned int irq,
void(*handler)(int, void *, struct pt_regs *),
```

unsigned long flags,
const char *dev_name,
void *dev_id);

irq	IRQ-nr
handler	Pointer to IRQ-handler
flags	Controls the behaviour of the interrupt
SA_INTERRUPT	Fast interrupt
SA_SHIRQ	
dev_name	The string passed to /proc/interrupts
dev_id	For shared interrupt lines

Returns:
0; Success
-EBUSY; IRQ already allocated
-EINVAL; IRQ outside allowed range

void free_irq(unsigned int, irq, void *dev_id);

What IRQ shall I use?

- Manuals
- Conventions
- Read a status bit or PCI config space
- Probe!!

unsigned long mask;
int irq;
mask = probe_irq_on(); /*Returns a mask of unused IRQ's*/
.
Generate an interrupt
.
irq=probe_irq_off(mask);
.
Fast interrupt
-SA_INTERRUPT
All interrupts disabled in the processor
Interrupt being serviced disabled in interrupt controller

Slow interrupt

All interrupts enabled in the processor
Interrupt being serviced disabled in interrupt controller

Rules

Is not run in an applications environment and cannot transport data to/from userspace
Are not allowed to sleep or wait for a semaphore
Keep the code short!!

Use of interrupts in drivers

Application calling read. No data ➔ sleeps
Data arrives and trigger an interrupt that wakes the read function

Enabling/Disabling interrupts

Enabling/Disabling all interrupts

```
Unsigned long flag;
.
save_flags(flags);
cli();
.              /* All Interrupts disabled */
.
restore_flags(flags);  /*Call save_flags &
                          restore_flags in same function!!*/
```

Enabling/Disabling one interrupt

```
void disable_irq( int irq);
void enable_irq( int irq);
```

Interrupts & Tasklets

Tasklets offers a way to divide interrupt tasks into two parts:
One short interrupthandler
A "working" part, run later as "common" code.

Tasklets

Executes in "interrupt" environment, in safe time (interrupts enabled) and never before the interrupt is completed.

Signaler i Linux

Avbrott

Stannar upp exekveringen av programmet, skickar kontrollen till en avbrottsfunktion som exekveras och som sedan (ev) återlämnar kontrollen till programmet

Två typer av avbrott:

Hårdvaruavbrott; Har vi redan behandlat, så dem lämnar vi nu därhän
Mjukvaruavbrott

Mjukvaruavbrott kallas i Linux/Unix ”Signal”

31 st (nr 1-31)
Alla har definierade namn som används, och som alla börjar på SIG...tex SIGKILL, SIGHUP, SIGABRT m.m.

Genereras på olika sätt:

Kommandot Kill från en användare
Hårdvaruavbrott som rapporteras till kärnan som sedan genererar en signal tex vid div med noll
Funktionen kill() i en process
Egna processen anropar funktioner som genererar avbrott. Tex abort()
Div. blandade mjukvarutillstånd, tex när man försöker skriva till en stängd pipa

Reaktion på en signal

Ignorera. Detta måste vi dock explicit skriva i vår kod. SIGKILL och SIGSTOP kan dock inte ignoreras
Default-händelse
Fånga upp signalen och kör en egen funktion

Exempel på signaler

6	SIGABRT Genereras av abort()	
14	SIGALRM	En timer genererar detta avbrott efter önskad tid
9	SIGKILL	Kan ej stängas av. Dödar processen
1	SIGHUP	Kan ta sig fram till en process utan terminal (tex en demon). Brukar användas till att få demoner att läsa in konfigfiler
10	SIGUSR1	Fri för egen användning
12	SIGUSR2	Fri för egen användning

Registrera egen signalhantering

```
void  (*signal(int signo, void (*func)(int)))(int)
```

Exempel exfil_1.c:

```
static void my_sigusr1_handler(int signo)
{
        do something;
        return;
}

int main(void)
{
        if(signal(SIGUSR1,my_sigusr1_handler)==SIGERR)
        {
                Error!!
        }
        for(;;);
}
```

exfil_2.c:

```
int main(void)
{
        .
        if(kill(pid, SIGUSR1) /*pid==exfil1's pid*/
        {
                Error sending signal
        }
}
```

alt:
[bl@mydator bl] kill –10 <pid för exfil_1>

Timers/Timing management

Two main kinds of timing measurement:
Provide the system with current time and date
Maintaining timers –Notify the system that certain interval of time has elapsed.

Maintains:
Updates the time elapsed since system startup
Updates Time & Date
Support the scheduler
Maintain statistics
Manages timers

4 clocks:

- Real Time Clock
- Time Stamp Counter, TSC
- Programable Interval Timer, PIT
- APICs, Advanced Programmable Interrupt Controllers

Real Time Clock
A CPU independent clock for deriving time & date.

Commonly connected to IRQ8.

Time Stamp Counter
64 bit register in the processor, incremented every clock signal.
1 GHz system increments the RTC every nanosecond.

PIT
Programmable device for generating interrupts, commonly at IRQ0 with 100 Hz.
Triggers all important time-keeping activities. 16 bits

APIC
32 bits. Similar to PIT's.

Application time calls

`time()`
Returns the number of elapsed secs since midnight 1/1 1970.

```
Time_t my_time;
my_time=time((time_t *)0);
```

gettimeofday()
Returns a struct timeval, containing elapsed secs since midnight 1/1 1970 and elapsed millisec in the last sec.

```
Struct timeval
{
        long tv_sec;
        long tv_usec;
}

struct timeval tv;

gettimeofday(&tv, NULL);
printf("Seks = %lu\n, tv.tv_sec);
printf("useks = %lu\n, tv.tv_usec);

clock_t clock(void);
```

Returns number of clockticks of CPU-time the program have used

```
Printf("Clock() ger %lu\n", clock());
```

```
Rdtsc(low,high);
Rdtscl(low);
```

Architecture dependent
Writes the value of TSC to the variables low & high

#include </usr/src/linux-x.x/include/asm/msr.h.
unsigned long low, high;
rdtsc(low,high);
rdtscl(low);
cycles_t get_cycles(void);

Kernel Space

do_gettimeofday(struct timeval *tv);

Compare with gettimeofday(…);

Jiffies

Incremented by IRQ0 interrupt service routine, which is triggered by the PIT.
Controls the scheduler and lots of other kernel activities.

Delays

#include <linux/delay>

void udelay(unsigned long usecs); Maintains precision up to 1 msec
void mdelay(unsigned long msecs);

Doesn’t allow other tasks to run!
And Hangs the system!!

Timers

Install timer handler function and connect it to a system signal.
Configure and set appropriate timing values
Start the timer.

```
Void timer_handler(int signum)
{
}

int main(…)
{
        struct sigaction sa;
        struct itimerval timer;

        memset(&sa, 0, sizeof(sa));
        as.sa_handler = &timer_handler;
        sigaction(SIGALRM, &sa, NULL);

        timer.it_value.tv_sec = 0;
        timer.it_value.tv_usec = 250000;

        timer.it_intervall.tv_sec = 0;
        timer.it_intervall.tv_usec = 250000;

        setitimer(ITIMER_REAL, &timer, NULL);
                /*ITIMER_REAL ➔ The process will be sent a
                SIGALRM signal */

        while(1);
}
```