Advanced Device Drivers

Alessandro Rubini, Universita' di Pavia, Italy


```
--
   __ o   La forza dei forti sta nel traversare le traversie con occhio sereno
  _`\<,                                                     (Paperino)
__( )/( )__      alessandro.rubini@linux.it    +39 382 529554
```


Device Driving

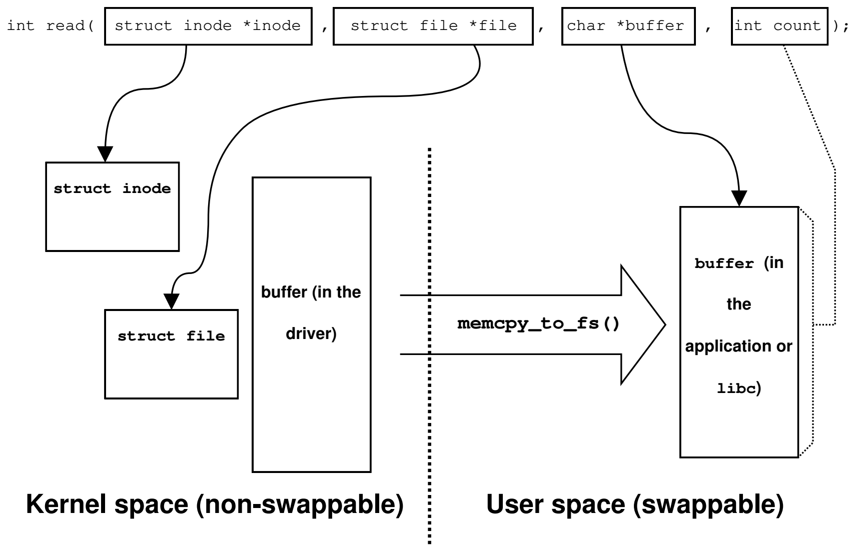
A driver exists on two different levels

Synchronous Level	Asynchronous Level
`open()`	Sleeping and Awakening
`close()`	Select
`read()`	Asynchronous Notification
`write()`	Interrupts
`ioctl()`	Bottom-Halves
	Timers

Synchronous Operations

```
struct file_operations {
        int (*lseek) (struct inode *, struct file *, off_t, int);
        int (*read) (struct inode *, struct file *, char *, int);
        int (*write) (struct inode *, struct file *, const char *
        int (*readdir) (struct inode *, struct file *, void *, fi
        int (*select) (struct inode *, struct file *, int, select
        int (*ioctl) (struct inode *, struct file *, unsigned int
        int (*mmap) (struct inode *, struct file *, struct vm_are
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
        int (*fasync) (struct inode *, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
};
```

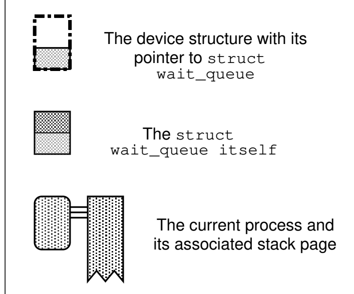
The single problem is data copying beween address spaces



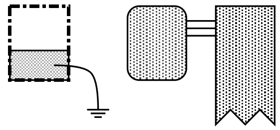
The software interface

```
extern void sleep_on(struct wait_queue ** p);
extern void interruptible_sleep_on(struct wait_queue ** p);
extern void select_wait(struct wait_queue ** wait_address,
                select_table * p);
extern void wake_up(struct wait_queue ** p);
extern void wake_up_interruptible(struct wait_queue ** p);
extern void wake_up_process(struct task_struct * tsk);
```

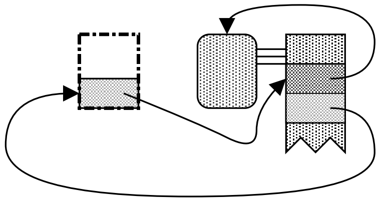
The most interesting part of a driver runs when processes sleep



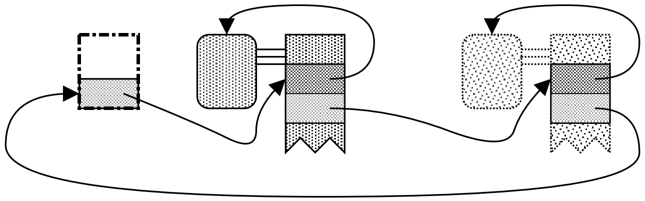
No process is sleeping on the queue



The current process is sleeping on the device's queue



Several processes are sleeping on the same queue

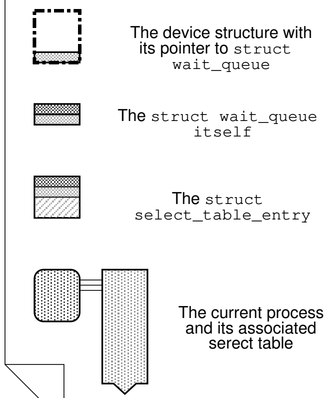


The select implementation

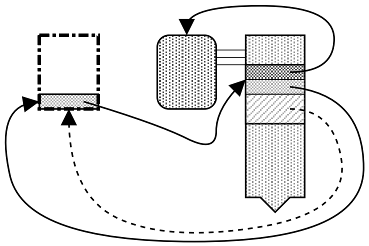
```
int generic_select (struct inode *inode, struct file *filp,
                          int sel_type, select_table *wait)
{
    struct generic_hw *hw = filp->private_data;

    switch (sel_type) {
        case SEL_IN: if (hw->rd_buffer.count) return 1;
        case SEL_OUT: if (hw->wr_buffer.room) return 1;
        case SEL_EX: return 0; /* no exceptions pending */
     }
     select_wait(&client->queue,wait);
     return 0;
}
```

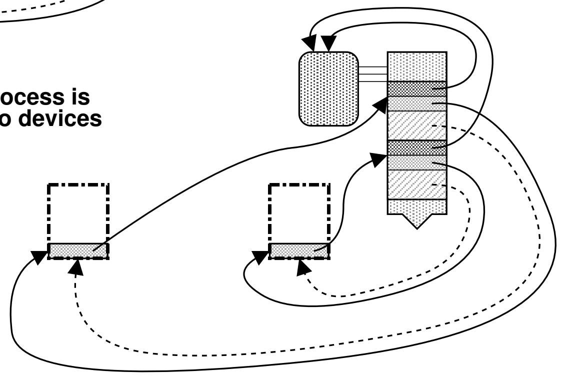
The tricky select implementation is hidden from the driver



The current process calls select for one device only



The current process is selecting on two devices



Asyncronous Input

```
struct generic_hw {
    /* ... */
    struct fasync_struct *async_queue;
};
```

During the System Calls

```
int generic_fasync (struct inode *inode, struct file *filp, int mode)
{
    struct generic_hw hw = filp->private_data;
    return fasync_helper(inode, filp, mode, &hw->async_queue);
}

void generic_close (struct inode *inode, struct file *filp)
{
     scull_p_fasync(inode, filp, 0);
     /* ... */
}
```

When Data Arrives

```
if (hw->async_queue) kill_fasync (hw->async_queue, SIGIO);
```

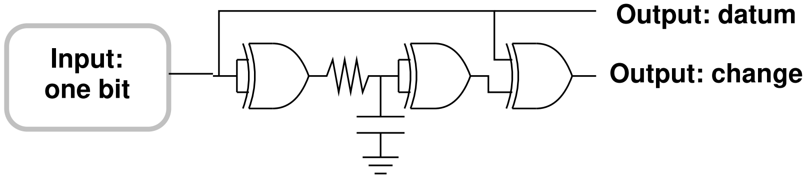
Although the kernel does it all, a driver method is needed

Asynchronous notification, actually is rarely *used*: it's not that easy.

```
signal(SIGIO, &input_handler);
fcntl(0, F_SETOWN, getpid());
oflags=fcntl(0, F_GETFL);
fcntl(0, F_SETFL, oflags | FASYNC);
```

Interrupt Handling

Although interrupt handling sits at the lowest level of abstraction, its implementation is quite trivial in Linux.



Let's try to handle an interrupt raised from the "change" bit above. Assume to connect to an input port.

```
void barcode_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    do_gettimeofday(&tv);

    /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */
    barcode_head += sprintf((char *)head,"%i,%06u.%06u\n", inb(INPORT)&1,
                    (int)(tv.tv_sec % 1000000), (int)(tv.tv_usec));
    if (barcode_head == barcode_buffer + PAGE_SIZE)
        barcode_head = barcode_buffer; /* wrap */

    wake_up_interruptible(&barcode_queue); /* awake any reading process */
}
```

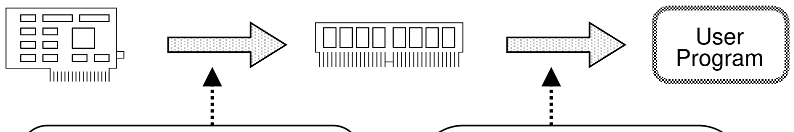
Handlers behave differently according to the following flags:

`SA_INTERRUPT` (be 'fast', default is 'slow')
`SA_SHIRQ` (share the line with another device)
`SA_SAMPLE_RANDOM` (contribute to the system's entropy pool).

Split Handlers

The main problem with handlers is their habit to do their job, and to do it well.

But the data path can be split in two steps



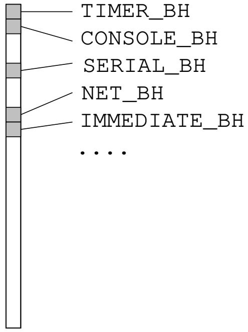
Retrieve data
Acknowledge it
Dispatch the bottom half

Make sanity checks
Collect statistics
Process line disciplines
Dispatch to processes
Notify async readers
Awake sleeping readers

There's a limited number of bottom-halves, and the top half must 'mark' them

The bottom half is run
- At return from system call
- At exit from a slow handler
- From the scheduler

Bottom halves exist for the most important tasks:




```
extern unsigned long bh_active;
extern unsigned long bh_mask;
extern void (*bh_base[32])(void);
extern void mark_bh(int nr)
extern void disable_bh(int nr)
extern void enable_bh(int nr)
```

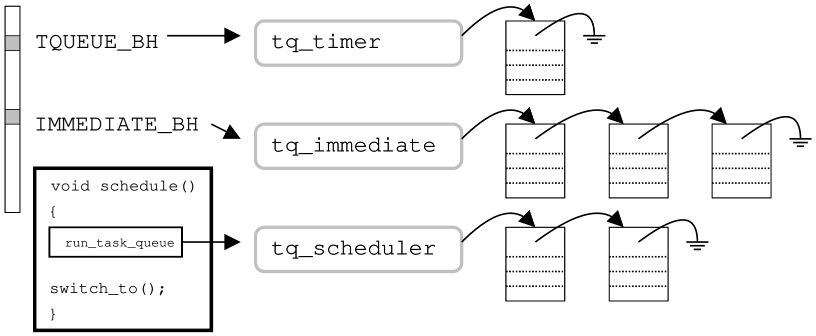
Using Task Queues

A task queue is an higher level tool than a bottom half: it's dynamic

```
struct tq_struct {
        struct tq_struct *next;
        int sync;
        void (*routine)(void *);
        void *data;   };

typedef struct tq_struct * task_queue;
extern void queue_task(struct tq_struct *bh_pointer,
                       task_queue *bh_list);
extern void run_task_queue(task_queue *list);

extern task_queue tq_timer, tq_immediate, tq_scheduler, tq_disk;
```




```
void barcode_fast_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   do_gettimeofday(tv_head); tv_head++;
   if (tv_head == (tv_data + NR_TIMEVAL) ) tv_head = tv_data; /* wrap */

   if (!barcode_count) {
       queue_task_irq_off(&barcode_task, &tq_immediate);
        mark_bh(IMMEDIATE_BH);
   }
   barcode_count++;
}


void barcode_bottom_half(void *unused)
{
   short_bh_count = 0;
   do {
       barcode_head += sprintf((char *)short_head,"%1,%06u.%06u\n",
           (int)(tv_tail->tv_sec % 1000000), (int)(tv_tail->tv_usec));
       if (barcode_head == barcode_buffer + PAGE_SIZE)
           barcode_head = barcode_buffer; /* wrap */
       tv_tail++;
       if (tv_tail == (tv_data + NR_TIMEVAL) )
           tv_tail = tv_data; /* wrap */
   } while (tv_tail != tv_head);
   wake_up_interruptible(&barcode_queue); /* awake any reading process */
}
```

Advanced Device Drivers UseLinux 1997, Anaheim, CA

Kernel Timers

A timer schedules a task for execution at a specific time in the future

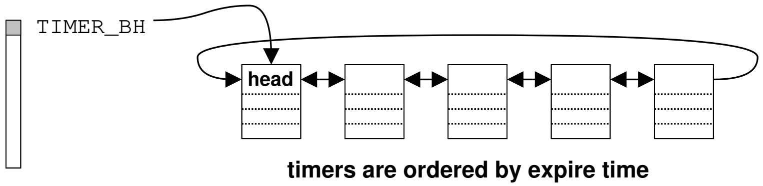
```
struct timer_list {
        struct timer_list *next;
        struct timer_list *prev;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

Timers can be used for
- error on timeout for devices
- error in timeout for processes
- motor-off or other 'positive' timeouts
- keyboard repetition for 'dumb' keyboards
- ....

```
extern void add_timer(struct timer_list * timer);
extern int  del_timer(struct timer_list * timer);

extern struct timer_list timer_head;
```

→ `current->timeout` is implemented using the timers



timers are ordered by expire time

```
int timely_open (struct inode *inode, struct file *filp)
{
    /* ... */
    del_timer(&dev->timer);
    dev->usage++;
    return 0;           /* success */
}

void timely_close (struct inode *inode, struct file *filp)
{
    /* ... */
    dev->usage--;
    if (!dev->usage) { /* die in 30 seconds */
        dev->timer.expires = jiffies + 30 * HZ;
        add_timer(&dev->timer);
    }
}
```

Kernel Threads

Kernel threads are processes that only live in the kernel space

USER	PID	%CPU	%MEM	SIZE	RSS	TTY	STAT	START	TIME	COMMAND
root	2	0.0	0.0	0	0	?	SW	12:22	0:00	(kflushd)
root	3	0.0	0.0	0	0	?	SW<	12:22	0:00	(kswapd)
root	72	0.0	0.0	0	0	?	SW	12:22	0:00	(kmouse)
root	533	0.0	0.0	0	0	13	SW	15:28	0:00	(nfsiod)
root	534	0.0	0.0	0	0	13	SW	15:28	0:00	(nfsiod)
root	535	0.0	0.0	0	0	13	SW	15:28	0:00	(nfsiod)
root	536	0.0	0.0	0	0	13	SW	15:28	0:00	(nfsiod)

```
#define __KERNEL__
#define __KERNEL_SYSCALLS__
#include <linux/unistd.h>
#include <asm/unistd.h>

extern pid_t kernel_thread(int (*fn)(void *),
               void *arg, unsigned long flags);
```

The following flags are supported:

```
CLONE_VM
CLONE_FS
CLONE_FILES
CLONE_SIGHAND
CLONE_PID
```

The need for a kernel thread arises when the programmer has to seriously work with kernel data structures. This can't be accomplished with task queues, timers or similar tools because their execution is bound to severe time constraints.

Conclusions

The Linux kernel offers a good deal of tools to efficiently handle the asynchronous part of device driving. This is much more than just reading and writing data or commands.

By using these tools one can easily handle real hardware and also create virtual devices: kernel code can change the process' view of the world.

An example: structure of the kmouse module