Linux Device Drivers – Interrupt Requests

Jernej Vičič

Overview

Introduction

- primitive devices can be managed only with I/O regions,
- most devices require a more complicated approach,
- devices cooperate with the environment:
  - disk rotates,
  - data is sent to remote locations,
  - tape is moved to the desired location,
- processor can not wait for slow devices,
- use interrupts.

Interrupts

- interrupt is a signal,
- triggered by hardware,
- when the processor needs attention,
- Linux handles interrupts the same as signals (user space),
- driver only registers an interrupt handler,
- the example code will "talk" with a parallel interface,
- extension of module *short*.

Preparing parallel port

- The parallel interface sends an interrupt when the electrical signal changes from *low-high* on the pin 10 (ACK bit),
- this is done by the printer when requesting data,
- interrupts are generated only when the bit 4 is on at the port 2,
- *short* makes the call *outb* at 0x37a,
- interrupt is triggered when the device changes pin 10:
  - from low to high state,
  - this can be done by connecting pins 9 and 10,
  - a piece of wire can be used,
  - can be connected to the printer.

Installation of an interrupt handler

- hardware generates interrupts,
- we need a program handler,
- the handler responds to interrupts,
- without the handler kernel only accepts interruptions,
- module requires *interrup channel* or *IRQ - Interrupt ReQuest*.

Installation of an interrupt handler

```
<linux/interrupt.h>

int request_irq(unsigned int irq,
irqreturn_t (*handler)(int, void *, struct pt_regs *),
unsigned long flags,const char *dev_name,
void *dev_id);

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

- *request_irq* – registers the handler,
- *free_irq* – frees the handler,

Installation of an interrupt handler

- *unsigned int irq* – number of requested interrupt,
- *irqreturn_t (*handler)(int, void *, struct pt_regs *)* – pointer to interrupt function,
- *unsigned long flags* – *bit mask*, that chooses options,
- *const char *dev_name* – string, written to */proc/interrupts* (owner),
- *void *dev_id* – pointer to shared interrupt lines.

Installation of an interrupt handler

- *SA_INTERRUPT* – *fast* handler, interrupts are masked,
- *SA_SHIRQ* – interrupt can be shared between devices,
- *SA_SAMPLE_RANDOM* – used to increase entropy for */dev/random* and */dev/urandom*.

Installation of an interrupt handler

```
if (short_irq >= 0) {
  result = request_irq(short_irq, short_interrupt,
    SA_INTERRUPT, "short", NULL);
  if (result) {
    printk(KERN_INFO "short: can't get assigned irq %i\n",
      short_irq);
    short_irq = -1;
  }
  else { /* pišemo na 4. bit drugih vrat */
    outb(0x10,short_base+2);
  }
}
```

- driver handler for *short*,
- the handler is *fast* type,
- does not support interrupts,
- does not increase entropy.

Interface /proc

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
CPU0 CPU1
0: 4848108 34 IO-APIC-edge timer
2: 0 0 XT-PIC cascade
8: 3 1 IO-APIC-edge rtc
10: 4335 1 IO-APIC-level aic7xxx
11: 8903 0 IO-APIC-level uhci_hcd
12: 49 1 IO-APIC-edge i8042
NMI: 0 0
LOC: 4848187 4848186
ERR: 0
MIS: 0
```

- output to */proc/interrupts*,
- two processors,
- columns:
  1. interrupt number.
  2. number of interrupts for CPU0.
  3. number of interrupts for CPU1.
  4. programmable interrupt controller (not interresting).

Interface /proc

```
more /proc/stat

cpu  12767861 277112 3399214 145240432 657031 15929 150065 0 0
cpu0 6295455 34237 1425503 69413239 550892 14165 7462 0 0
cpu1 6472406 242874 1973710 75827193 106139 1764 142602 0 0
intr 645970816 2633 126790 0 0 3 0 0 0 1 0 0 ...
ctxt 2162516636
btime 1331725015
processes 79217
procs_running 2
procs_blocked 0
softirq 730115300 0 311183802 2953860 4968292 3034293 0 898854 103466656 23267 303586276
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0 0
```

- interrupt statistics.

Handler implementation

- can not transfer data to/from the user space,
- should not be put to sleep (call *event_wait*),
- memory can only be allocated with *GFP_ATOMIC*,
- can not run *schedule*,
- Purpose of the handler:
  - send information to device,
  - reading from device,
  - writing to the device,

Handler implementation

- execution time is critical,
- if it takes longer, use:
  - tasklet,
  - workqueue,
- the execution will be executed at an appropriate time,

Handler implementation - SHORT

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  struct timeval tv;
  int written;
  do_gettimeofday(&tv);
  /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */
  written = sprintf((char *)short_head,"%08u.%06u\n",
    (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
  BUG_ON(written != 16);
  short_incr_bp(&short_head, written);
  wake_up_interruptible(&short_queue); /* awake any reading process */
  return IRQ_HANDLED;
}
```

Handler implementation

- call function *to_gettimeofday*,
- displays the current time in a circular buffer,
- wakes up all sleeping processes (data are coming),
- represents a typical interrupt handler task.
- calls function *short_incr_bp*:

Handler implementation

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
  unsigned long new = *index + delta;
  barrier( ); /* Don't optimize these two together */
  *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

- pointer is wrapped in a circular buffer,
- never uses the wrong location (checks),
- uses a barrier (to prevent optimization),

Handler implementation, no barrier

- compiler could optimize code:
- *new* = **index*,
- for a moment could be a wrong value in *index*,
- the other thread could use the wrong value.
- now we can use the circular buffer without locking.

Handler implementation

- device file for reading the described buffer is */dev/shortint*,
- it is not described in the previous chapter,
- a special file showing the use of interrupts,
- writing to a file triggers an interrupt for each other byte,
- reading shows when an interrupt occurred,

Handler implementation

- connect pins 9 and 10,
- we can set the most important bit (high bit) of the parallel data port,
- write data to */dev/short0* or */dev/shortint*,

Handler implementation

```
ssize_t short_i_read (struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
  int count0;
  DEFINE_WAIT(wait);
  while (short_head = = short_tail) {
    prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
    if (short_head = = short_tail)
      schedule( );
    finish_wait(&short_queue, &wait);
    if (signal_pending (current)) /* a signal arrived */
      return -ERESTARTSYS; /* tell the fs layer to handle it */
  }
  /* count0 is the number of readable data bytes */
  count0 = short_head - short_tail;
  if (count0 < 0) /* wrapped */
    count0 = short_buffer + PAGE_SIZE - short_tail;
  if (count0 < count) count = count0;
  if (copy_to_user(buf, (char *)short_tail, count))
    return -EFAULT;
  short_incr_bp (&short_tail, count);
  return count;
}
```

Handler implementation

```
ssize_t short_i_write (struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
  int written = 0, odd = *f_pos & 1;
  unsigned long port = short_base; /* output to the parallel data latch */
  void *address = (void *) short_base;
  if (use_mem) {
    while (written < count)
    iowrite8(0xff * ((++written + odd) & 1), address);
  } else {
    while (written < count)
    outb(0xff * ((++written + odd) & 1), port);
  }
  *f_pos += count;
  return written;
}
```

Handler implementation

- */dev/shortprint* allows you to "talk" to the printer,
- slightly modified example described,
- it is not necessary to connect the pins 9 and 10,
- writing uses a buffer to store data for printing,
- reading gets the time the printer needs to read one character.

Time synchronization

- Often handlers must perform long-lasting operations,
- handlers must end as soon as possible (other interruptions are blocked),
- these two requests are exclusive.
- Linux solves this problem by dividing the handler into two parts.

Division in two parts

- two halves,
- not only Linux,
- *top half*:
  - routine that is actually called at the interrupt,
  - this is registered with function *request_irq*,
- *bottom half*:
  - routine that is scheduled by first routine (put into queue),
  - it is executed at an appropriate time,
- interrupts are not masked at bottom half,

Division in two parts

- typical scenario for the top half:
  - the top half saves the data from the device to the buffer,
  - allocates its lower half to later implementation,
  - top half ends,
  - this happens very fast.

Division in two parts

- typical scenario for the bottom half:
  - it is executed when the time is right,
  - enabled interrupts,
  - performs all necessary work:
    - wakes up sleeping processes,
    - runs other I/O operations,
    - ....

Division in two parts

- almost every handler is such,
- example of a network driver:
  - network interface logs a new packet,
  - the handler only retrieves the data,
  - moves them to "protocol layer",
  - the processing is done in the bottom half,
  - is carried out at an appropriate time,
  - a new packet might come in the middle.

Division in two parts

- Linux has two mechanisms:
  - *tasklets*:
    - fast,
    - still need atomic code,
    - mostly used,
  - *workqueues*:
    - code can be put to sleep,
    - slower execution

Tasklets

- special features,
- run in the context of interrupts,
- running in "reasonable time" = *system-determined safe time*,
- tasklet can be triggered several times, it will only be executed once,
- tasklet is never executed in parallel with itself,
- tasklets can be run in parallel with other tasklets (on multiple processors).

Tasklets

- tasklet is executed on the same CPU as the handler,
- never in parallel,

Tasklets

```
DECLARE_TASKLET(name, function, data);
```

- declare tasklets,
- *name* – tasklet name,
- *function* – function, called by (tasklet),
- *data* – unsigned long function parameter ,
- example in *short*:

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

Tasklets

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  do_gettimeofday((struct timeval *) tv_head); /* cast to stop 'volatile' warning
  */
  short_incr_tv(&tv_head);
  tasklet_schedule(&short_tasklet);
  short_wq_count++; /* record that an interrupt arrived */
  return IRQ_HANDLED;
}
```

- this handler is used if we start *short* with the parameter *tasklet* = *1*,
- a second handler is installed,
- *tasklet_schedule* - this function is used when scheduling tasklets,
- tasklet routine *short_do_tasklet* is executed when the system wants it,

Tasklets

```
void short_do_tasklet (unsigned long unused)
{
  int savecount = short_wq_count, written;
  short_wq_count = 0; /* we have already been removed from the queue */
  /*
  * The bottom half reads the tv array, filled by the top half,
  * and prints it to the circular text buffer, which is then consumed
  * by reading processes
  */
  /* First write the number of interrupts that occurred before this bh */
  written = sprintf((char *)short_head,"bh after %6i\n",savecount);
  short_incr_bp(&short_head, written);
  /*
  * Then, write the time values. Write exactly 16 bytes at a time,
  * so it aligns with PAGE_SIZE
  */
  do {
    written = sprintf((char *)short_head,"%08u.%06u\n",
    (int)(tv_tail->tv_sec % 100000000),
    (int)(tv_tail->tv_usec));
    short_incr_bp(&short_head, written);
    short_incr_tv(&tv_tail);
  } while (tv_tail != tv_head);
  wake_up_interruptible(&short_queue); /* awake any reading process */
}
```

Workqueues

- workqueues call the function,
- special context – *worker process*,
- functions can sleep,
- we can not copy the data to/from the user space,
- except in special ways (it does not matter for us),

Workqueues

```
static struct work_struct short_wq;
/* this line is in short_init( ) */
INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

- use the driver with parameter $wq = 1$,
- *work_struct* – this structure is used for workqueues.

Workqueues

```
irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  /* Grab the current time information. */
  do_gettimeofday((struct timeval *) tv_head);
  short_incr_tv(&tv_head);
  /* Queue the bh. Don't worry about multiple enqueueing */
  schedule_work(&short_wq);
  short_wq_count++; /* record that an interrupt arrived */
  return IRQ_HANDLED;
}
```

- new handler,
- similar to tasklet, calls *schedule_work* for scheduling lower part.