Today's class

Cleanup: Interrupt Bottom Halves, Intelligent Buses
New: Loadable kernel modules
Homework: look at device driver code and device tree

Next week

DMA controllers and hardware data direct copying to RAM
Linux booting process finish – **bootloader and devices**, root file system and the init_kernel function

Interrupt Bottom Halves

Top Half vs. Bottom Half

These two goals of an interrupt handler conflict with one another

- Execute quickly
- Perform a large amount of work

So the processing of interrupts is split into two parts, or halves

Top half

The interrupt handler is the top half. The top half is run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware.

Bottom half

Work that can be performed later is deferred until the bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled.

Example using network card

- When network cards receive packets from the network, the network cards immediately issue an interrupt. This optimizes network throughput and latency and avoids timeouts.
- The kernel responds by executing the network card's registered interrupt.
- The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and readies the network card for more packets. These jobs are the important, time-critical, and hardware-specific work.
- The rest of the processing and handling of the packets occurs later, in the bottom half

Again many mechanisms for deferred work/ bottom halves

Bottom Half	Status	
BH	Removed in 2.5	
Task queues	Removed in 2.5	
Softirq	Available since 2.3	Networking and block devices
Tasklet	Available since 2.3	
Work queues	Available since 2.5	

- Interrupts are enabled in bottom half (unlike interrupt handler)
- Softirq and tasklets cannot sleep
- Work queues can sleep
- Softirq needs proper locking (as different processors can execute same softirq code)
- Softirqs are statically allocated at compile time, tasklets can be dynamically created

Softirq <kernel/softirq.c>

- static struct softirq_action softirq_vec[NR_SOFTIRQS];
- 32 is the limit, only nine exist

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
TASKLET_SOFTIRQ	5	Normal priority tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_SOFTIRQ	7	High-resolution timers
RCU_SOFTIRQ	8	RCU locking

Handling softirq

- Assign a sofirq index

Handling softirq

- Assign a sofirq index
- **Register handler:** the softirq handler is registered at run-time via open_softirq(), which takes two parameters: the softirq's index and its handler function.The networking subsystem, for example, registers its softirqs like this, in net/core/dev.c :
  - open_softirq(NET_TX_SOFTIRQ, net_tx_action);
  - open_softirq(NET_RX_SOFTIRQ, net_rx_action);

Handling softirq

- Assign a sofirq index
- **Register handler:** the softirq handler is registered at run-time via open_softirq(), which takes two parameters: the softirq's index and its handler function.The networking subsystem, for example, registers its softirqs like this, in net/core/dev.c :
  - open_softirq(NET_TX_SOFTIRQ, net_tx_action);
  - open_softirq(NET_RX_SOFTIRQ, net_rx_action);
- A registered softirq must be marked before it will execute.This is called **raising the softirq**. Usually, an interrupt handler marks its softirq for execution before returning. For example, the networking subsystem would call,
  - raise_softirq(NET_TX_SOFTIRQ);

Handling softirq

- Assign a sofirq index
- **Register handler:** the softirq handler is registered at run-time via open_softirq(), which takes two parameters: the softirq’s index and its handler function.The networking subsystem, for example, registers its softirqs like this, in net/core/dev.c :
  - open_softirq(NET_TX_SOFTIRQ, net_tx_action);
  - open_softirq(NET_RX_SOFTIRQ, net_rx_action);
- A registered softirq must be marked before it will execute.This is called **raising the softirq**. Usually, an interrupt handler marks its softirq for execution before returning. For example, the networking subsystem would call,
  - raise_softirq(NET_TX_SOFTIRQ);
- Then, at a suitable time, the softirq runs. Pending softirqs are checked for and executed in the following places:
  - In the return from hardware interrupt code path
  - In the ksoftirqd kernel thread
  - In any code that explicitly checks for and executes pending softirqs, such as the net-working subsystem

Kernel threads run softirq, tasklets

- Kernel does not immediately process reactivated softirqs. Instead, if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load.

Kernel threads run softirq, tasklets

- Kernel does not immediately process reactivated softirqs. Instead, if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load.
- The kernel threads run with the lowest possible priority (nice value of 19), which ensures they do not run in lieu of anything important.
- This concession prevents heavy softirq activity from completely starving user-space of processor time. Conversely, it also ensures that “excess” softirqs do run eventually.

Kernel threads run softirq, tasklets

- Kernel does not immediately process reactivated softirqs. Instead, if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load.
- The kernel threads run with the lowest possible priority (nice value of 19), which ensures they do not run in lieu of anything important.
- This concession prevents heavy softirq activity from completely starving user-space of processor time. Conversely, it also ensures that “excess” softirqs do run eventually.
- Finally, this solution has the added property that on an idle system the softirqs are handled rather quickly because the kernel threads will schedule immediately.
- There is one thread per processor.The threads are each named ksoftirqd/n where n is the processor number. On a two-processor system, you would have ksoftirqd/0 and ksoftirqd/1 . Having a thread on each processor ensures an idle processor, if available, can always service softirqs.

Kernel threads run softirq, tasklets

- Kernel does not immediately process reactivated softirqs. Instead, if the number of softirqs grows excessive, the kernel wakes up a family of kernel threads to handle the load.
- The kernel threads run with the lowest possible priority (nice value of 19), which ensures they do not run in lieu of anything important.
- This concession prevents heavy softirq activity from completely starving user-space of processor time. Conversely, it also ensures that “excess” softirqs do run eventually.
- Finally, this solution has the added property that on an idle system the softirqs are handled rather quickly because the kernel threads will schedule immediately.
- There is one thread per processor.The threads are each named ksoftirqd/n where n is the processor number. On a two-processor system, you would have ksoftirqd/0 and ksoftirqd/1 . Having a thread on each processor ensures an idle processor, if available, can always service softirqs.

```
rijurekha@rijurekha-Inspiron-5567:~$ ps -ef | grep softirq
root         3     2  0 Aug06 ?        00:00:05 [ksoftirqd/0]
root        13     2  0 Aug06 ?        00:00:03 [ksoftirqd/1]
root        18     2  0 Aug06 ?        00:00:01 [ksoftirqd/2]
root        23     2  0 Aug06 ?        00:00:01 [ksoftirqd/3]
rijurek+  2165 19975  0 08:37 pts/6    00:00:00 grep --color=auto softirq
```

Checking for pending tasks

- Regardless of the method of invocation, softirq execution occurs in __do_softirq(), which is invoked by do_softirq()

```
u32 pending;

pending = local_softirq_pending();
if (pending) {
    struct softirq_action *h;

    /* reset the pending bitmask */
    set_softirq_pending(0);

    h = softirq_vec;
    do {
        if (pending & 1)
            h->action(h);
        h++;
        pending >>= 1;
    } while (pending);
}
```

Work Queues

- If we need a schedulable entity to perform bottom-half processing, we need work queues.
- They are the only bottom-half mechanisms that run in process context, so the only ones that can sleep.

Work Queues

- If we need a schedulable entity to perform bottom-half processing, we need work queues.
- They are the only bottom-half mechanisms that run in process context, so the only ones that can sleep.
- Useful for situations in which we
  - need to allocate a lot of memory
  - obtain a semaphore
  - perform block I/O
- In its most basic form, the work queue subsystem is an interface for creating kernel threads to handle work queued from elsewhere.These kernel threads are called **worker threads**.

Running worker threads in Riju's machine

```
rijurekha@rijurekha-Inspiron-5567:~$ ps -ef | grep worker
root         5     2  0 Aug06 ?        00:00:00 [kworker/0:0H]
root        15     2  0 Aug06 ?        00:00:00 [kworker/1:0H]
root        20     2  0 Aug06 ?        00:00:00 [kworker/2:0H]
root        25     2  0 Aug06 ?        00:00:00 [kworker/3:0H]
root       169     2  0 Aug06 ?        00:00:01 [kworker/0:1H]
root       171     2  0 Aug06 ?        00:00:01 [kworker/2:1H]
root       205     2  0 Aug06 ?        00:00:01 [kworker/1:1H]
root       379     2  0 Aug06 ?        00:00:05 [kworker/3:1H]
root      1044     2  0 07:41 ?        00:00:01 [kworker/0:1]
root      1279     2  0 07:55 ?        00:00:00 [kworker/2:1]
root      1366     2  0 08:02 ?        00:00:00 [kworker/3:0]
root      1473     2  0 08:08 ?        00:00:00 [kworker/u8:0]
root      1741     2  0 08:17 ?        00:00:00 [kworker/1:0]
root      1797     2  0 08:21 ?        00:00:00 [kworker/u8:1]
root      1823     2  0 08:22 ?        00:00:00 [kworker/1:2]
root      1877     2  0 08:26 ?        00:00:00 [kworker/3:1]
root      1878     2  0 08:26 ?        00:00:00 [kworker/2:2]
root      1895     2  0 08:26 ?        00:00:00 [kworker/u8:2]
root      1997     2  0 08:30 ?        00:00:00 [kworker/0:2]
root      2105     2  0 08:34 ?        00:00:00 [kworker/3:2]
root      2135     2  0 08:35 ?        00:00:00 [kworker/0:0]
root      2158     2  0 08:37 ?        00:00:00 [kworker/u8:3]
rijurek+  2169 19975  0 08:37 pts/6    00:00:00 grep --color=auto worker
```

Choosing among mechanisms? -- policy

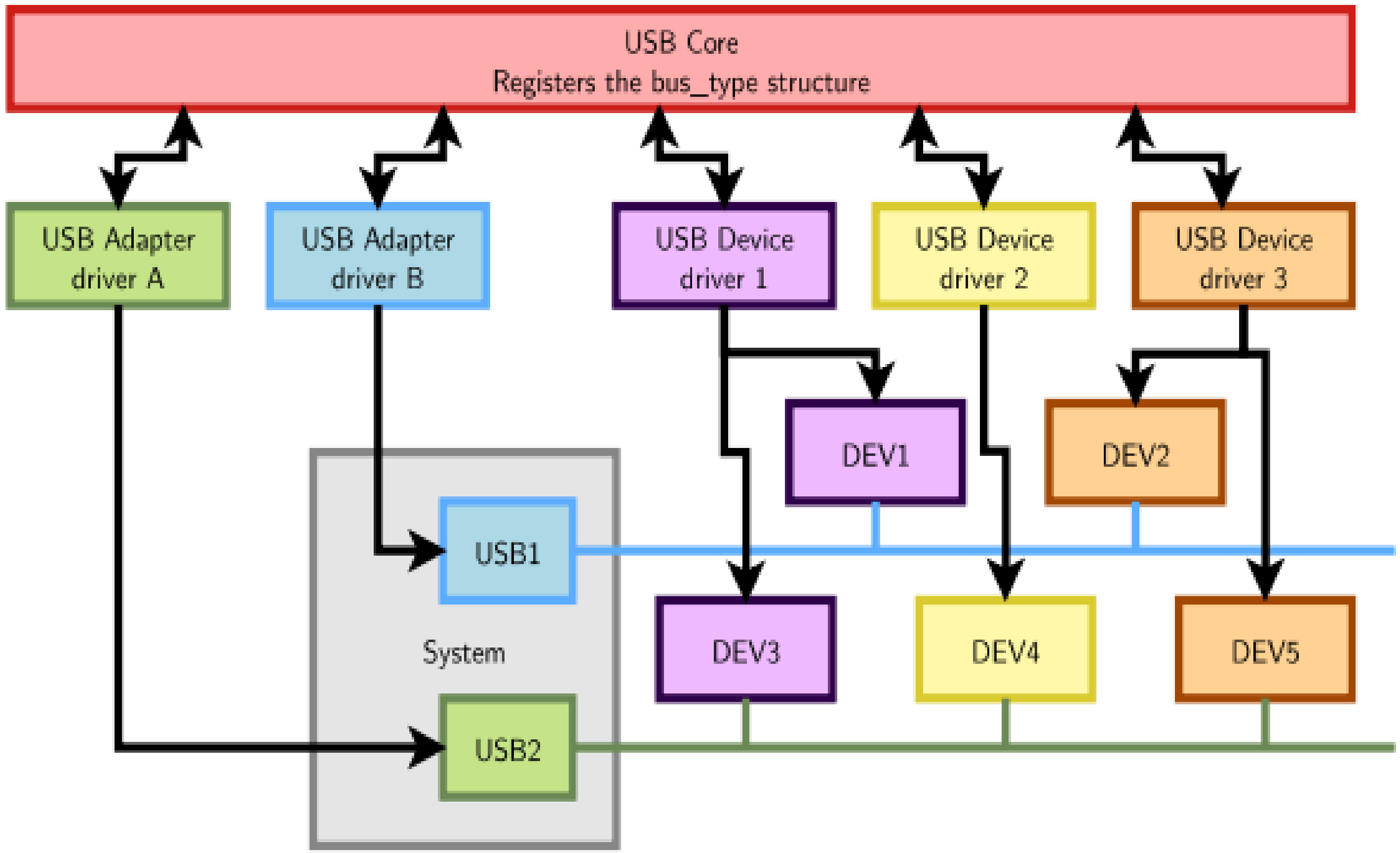
- If the deferred task needs to block (kmalloc, user space data copy), work queue is the only option
- For everything else, tasklets are good (dynamic allocation, need not worry about synchronization)
- For highly time critical tasks softirq-s are used, as same sofirq can run on different processors improving concurency (driver writer should take care of synchronization issues).

Intelligent buses that detect non-platform devices

Bus core driver for non-platform devices (detectable)

- E.g. USB or PCI
- Example: USB. Implemented in drivers/usb/core/
  - Creates and registers the bus_type structure
  - Provides an API to register and implement adapter drivers (here USB controllers), able to detect the connected devices and allowing to communicate with them.
  - Provides an API to register and implement device drivers (here USB device drivers)
  - Matches the device drivers against the devices detected by the adapter drivers.
  - Defines driver and device specific structures, here mainly struct usb_driver and struct usb_interface

A high level USB controller driver



A single driver for compatible devices, though connected to buses with different controllers.

Device Driver

Need to register supported devices to the bus core.

Example: drivers/net/usb/rtl8150.c

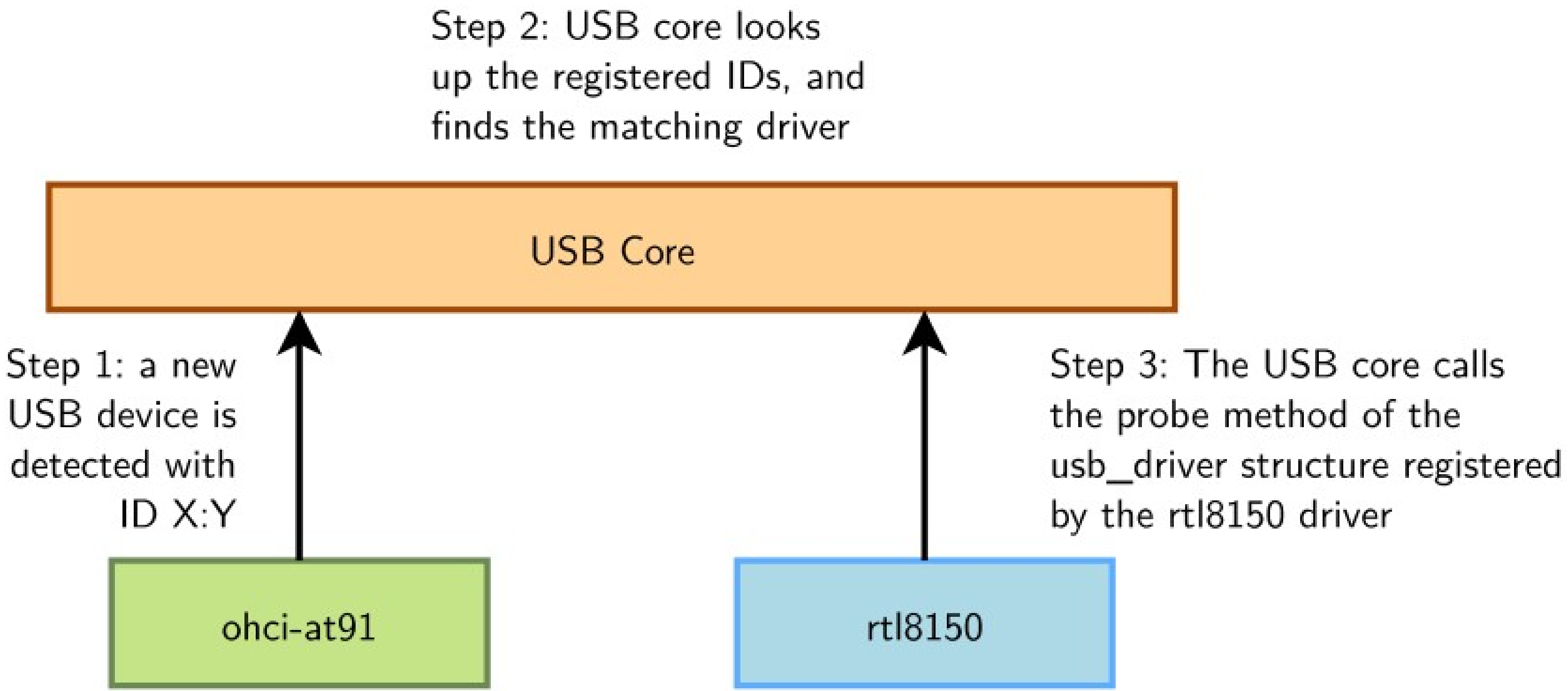
```
static struct usb_device_id rtl8150_table[] =
{{ USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150) },
{ USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX) },
{ USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR) },
{ USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX) },[…]
{}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Device Driver (contd.)

Need to register hooks to manage devices (newly detected or removed ones), as well as to react to power management events (suspend and resume)

```
static struct usb_driver rtl8150_driver = {
.name = "rtl8150",
.probe = rtl8150_probe,
.disconnect = rtl8150_disconnect,
.id_table = rtl8150_table,
.suspend = rtl8150_suspend,
.resume = rtl8150_resume
};
```

When a device is detected on bus



Loadable Kernel Modules

What is it, why useful?

- Software component which can be added to the memory image of the Kernel while it is already running.
  - The kernel does not need to be recompiled to add new software facilities
  - They are also used to develop new parts of the Kernel that can be then integrated in the final image once stable
  - They are also used to tailor the start up of a kernel configuration, depending on specific needs

Module Code hello.c

```c
#include <linux/init.h>             // Macros used to mark up functions e.g., __init __exit
#include <linux/module.h>           // Core header for loading LKMs into the kernel
#include <linux/kernel.h>           // Contains types, macros, functions for the kernel

MODULE_LICENSE("GPL");              ///< The license type -- this affects runtime behavior
MODULE_AUTHOR("Derek Molloy");      ///< The author -- visible when you use modinfo
MODULE_DESCRIPTION("A simple Linux driver for the BBB.");  ///< The description -- see modinfo
MODULE_VERSION("0.1");              ///< The version of the module

static char *name = "world";        ///< An example LKM argument -- default value is "world"
module_param(name, charp, S_IRUGO); ///< Param desc. charp = char ptr, S_IRUGO can be read/not changed
MODULE_PARM_DESC(name, "The name to display in /var/log/kern.log");  ///< parameter description

/** @brief The LKM initialization function
 *  The static keyword restricts the visibility of the function to within this C file. The __init
 *  macro means that for a built-in driver (not a LKM) the function is only used at initialization
 *  time and that it can be discarded and its memory freed up after that point.
 *  @return returns 0 if successful
 */
static int __init helloBBB_init(void){
   printk(KERN_INFO "EBB: Hello %s from the BBB LKM!\n", name);
   return 0;
}

/** @brief The LKM cleanup function
 *  Similar to the initialization function, it is static. The __exit macro notifies that if this
 *  code is used for a built-in driver (not a LKM) that this function is not required.
 */
static void __exit helloBBB_exit(void){
   printk(KERN_INFO "EBB: Goodbye %s from the BBB LKM!\n", name);
}

/** @brief A module must use the module_init() module_exit() macros from linux/init.h, which
 *  identify the initialization function at insertion time and the cleanup function (as
 *  listed above)
 */
module_init(helloBBB_init);
module_exit(helloBBB_exit);
```

printk messages in Riju's machine

```
rijurekha@rijurekha-Inspiron-5567:~$ tail -f /var/log/kern.log
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.148023] wlan0: RX AssocResp from d0:04:01:5f:bb:be (capab=0x431 status=0 aid=5)
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.149635] wlan0: associated
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.149784] IPv6: ADDRCONF(NETDEV_CHANGE): wlan0: link becomes ready
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.240371] wlan0: deauthenticating from d0:04:01:5f:bb:be by local choice (Reason: 2=PREV_AUTH_NOT_V
ALID)
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.254500] wlan0: authenticate with d0:04:01:5f:bb:be
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.260875] wlan0: send auth to d0:04:01:5f:bb:be (try 1/3)
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.263446] wlan0: authenticated
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.267385] wlan0: associate with d0:04:01:5f:bb:be (try 1/3)
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.279236] wlan0: RX AssocResp from d0:04:01:5f:bb:be (capab=0x431 status=0 aid=6)
Aug 11 07:04:11 rijurekha-Inspiron-5567 kernel: [202702.282654] wlan0: associated
```

Dynamically loading and removing modules

- A module is loaded by the administrator via the shell command **insmod**
- It takes as a parameter the path to the object file generated when compiling the module
- It can also be used to pass **parameters (variable=value)**
  - These are not passed as actual function parameters, but as initial values of global variables declared in the module source code
  - **sudo insmod hello.ko name=Derek** instead of "Hello World", "Hello Derek" will be printed
- A module is unloaded via the shell command **rmmod**
- We can also use **modprobe**, which by default looks for the actual module in the directory /lib/modules/$(uname –r)

Reference Counters

- The Kernel keeps a reference counter for each loaded LKM
- If the reference counter is greater than zero, then the module is locked
  - This means that there are processes in the system which rely on facilities exposed by the module
  - If not forced, unloading of the module fails

```
rijurekha@rijurekha-Inspiron-5567:~$ sudo lsmod
[sudo] password for rijurekha:
Module                  Size  Used by
drbg                   28672  1
ansi_cprng             16384  0
ctr                    16384  0
ccm                    20480  0
cmac                   16384  2
bnep                   20480  2
rfcomm                 69632  8
binfmt_misc            20480  1
snd_hda_codec_hdmi     53248  1
dell_led               16384  1
snd_hda_codec_realtek    90112  1
```

Today's class

Cleanup: Interrupt Bottom Halves, Intelligent Buses
New: Loadable kernel modules
Homework: look at device driver code and device tree

Next week

DMA controllers and hardware data direct copying to RAM
Linux booting process finish – **bootloader and devices**, root file system and the init_kernel function