Dive into HelenOS Device Drivers

Jiří Svoboda

Who am I?

- Jiří Svoboda
- Joined HelenOS in 2008 (master thesis)
- Day job: Sustaining Engineer
- Works on HelenOS in spare time
- Areas: debugging framework, input stack, block drivers, device driver framework, console, applications, networking

Drivers Available in HelenOS

- USB: UHCI, OHCI, HID, mass storage
- Network: Intel PRO/1000, NE2000, RTL8139
- ATA/PI disk / CD-ROM
- Legacy I/O (PS/2, CUDA, ...)

Agenda

- Driver requirements
- Programmed I/O and DMA in user space
- Level interrupts, tasklets
- Cooperation of user-space and kernel drivers
- DDF and Device Manager
- Exposing driver services, Location Service

Driver Requirements

HelenOS

- CPU architecture independence
- Platform independence
- Compositionality
- Automatic enumeration
- Hot-plug and unplug

Programmed I/O

- Kernel does not have to be in I/O path
  - Memory-mapped
    - simply map into task address space
  - Separate I/O space (ia32 & amd64)
    - I/O Permission Bitmap (part of TSS)
    - Lazy loading, in similar fashion to FPU context
- Endianness
  - host2uint{8|16|32}_t_{l|b}e()
  - uint{8|16|32}_t_{b|l}e2host()
- Modeling registers with C structs & unions
  - Beware __attribute__((packed))

Programmed I/O

- #include <ddi.h>
- pio_enable(void *pio_addr, size_t size, void **use_addr)
- physmem_map(void *pa, void *va, unsigned long pages, int flags)
- uint{8|16|32}_t pio_read(ioport{8|16|32}_t)
- pio_write_{8|16|32}(ioport{8|16|32}_t *port, uint{8|16|32}_t val)

DMA

- First-party: Allocate and map physical memory
  - physically contiguous
  - constraints (address width, alignment)
  - need support in physical memory allocator
  - mapped to driver address space
  - device programmed in device-specific manner
- int dmamem_map_anonymous(size_t size, unsigned int map_flags, unsigned int flags, void **phys, void **virt)

DMA

HelenOS

- Third party: DMA controller (in addition)
  - allocate DMA channel
  - program DMA channel (physical address, length)

Interrupt Handling

HelenOS

- Interrupt = (part of) mechanism to deliver signal (event) from device to device driver
- Could potentially transit several buses/controllers
  - each could affect interrupt number
  - each may require some setup, clearing, etc.
- Kernel delivers to user space in form of IPC message
- Problems
  - Level interrupts, Shared interrupts

Interrupt Handling

HelenOS

- int register_irq(int inr, int devno, int method, irq_code_t *ucode)
- int unregister_irq(int inr, int devno)

Tasklets

- Solution to problem of level interrupts
- Computational core provided by driver
- Interpreted language (simple instruction code)
  - Input/Output
  - Bit test
  - Predicate
  - Claim interrupt
- Executed in interrupt context
- Limited comp. strength (no backward jumps)

Tasklets

```
static irq_cmd_t i8042_cmds[] = {
	{
		.cmd = CMD_PIO_READ_8,
		.addr = NULL,   /* will be patched in run-time */
		.dstarg = 1
	},
	{
		.cmd = CMD_BTEST,
		.value = i8042_OUTPUT_FULL,
		.srcarg = 1,
		.dstarg = 3
	},
	{
		.cmd = CMD_PREDICATE,
		.value = 2,
		.srcarg = 3
	},
	{
		.cmd = CMD_ACCEPT
	}
};
```

```
static irq_code_t i8042_code = {
	sizeof(i8042_cmds) / sizeof(irq_cmd_t),
	i8042_cmds
};
```

Device Handover

HelenOS

- Kernel has some simple drivers
  - Frame buffer, keyboard/serial console, interrupt controller
  - For historical and debugging purposes
- Need handover between kernel and u. space
  - during boot when user-space console comes up
  - when switching to kernel console and back

Device Handoff

HelenOS

- hash_table_t irq_kernel_hash_table
- hash_table_t irq_uspace_hash_table

- irq_dispatch_and_lock()

- input_yield()

DDF/Device Manager

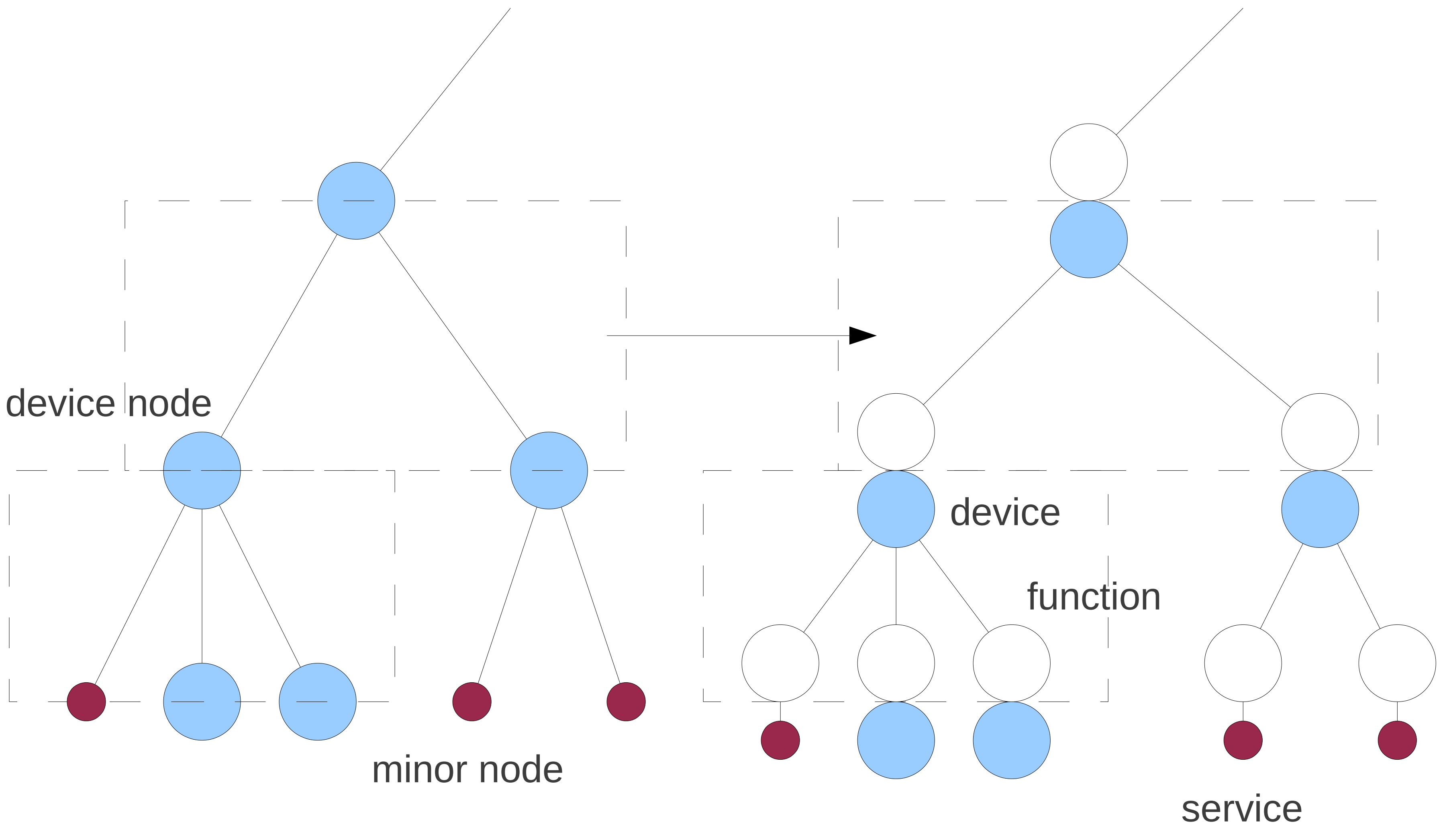
- libdrv – interface for driver
- Driver implements entry points
- Driver calls DDF functions
- Enumeration
- Automatic driver start
- Hot plug and unplug
- Command-line administration (devctl)

Device Model

HelenOS

- Cosmetic modification of classical device tree
- Split node into *device* and *function*
  - Driver instance *attaches to* a device
  - Driver instance *provides* one or more functions
- Functions are *inner* or *exposed*
  - Inner function – for attaching child drivers
  - Exposed function – served to external clients

Device Model



DDF/Device Manager

HelenOS

Driver entry points

- int (*dev_add)(ddf_dev_t *dev)
- int (*dev_remove)(ddf_dev_t *dev)
- int (*fun_online)(ddf_fun_t *fun)
- int (*fun_offline)(ddf_fun_t *fun)

DDF/Device Manager

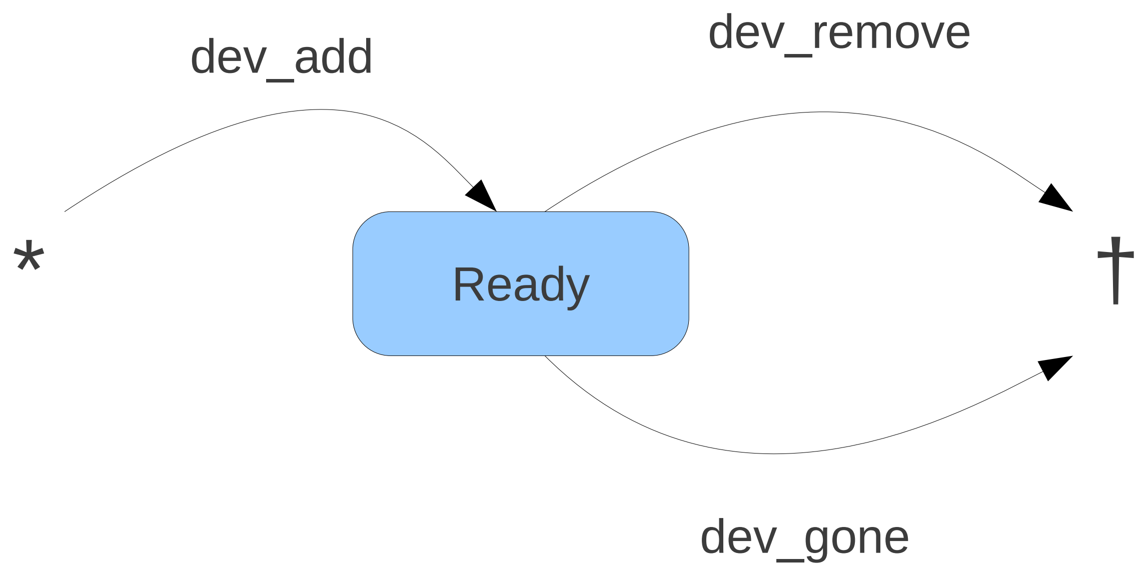
HelenOS

DDF functions

- ddf_fun_{create|destroy}()
  - driver provides hooks to handle incoming requests
- ddf_fun_{bind|unbind}()
- ddf_fun_add_match_id()
- ddf_fun_add_to_
- ddf_fun_{online|offline}()
- Connect to parent device

Device Life Cycle

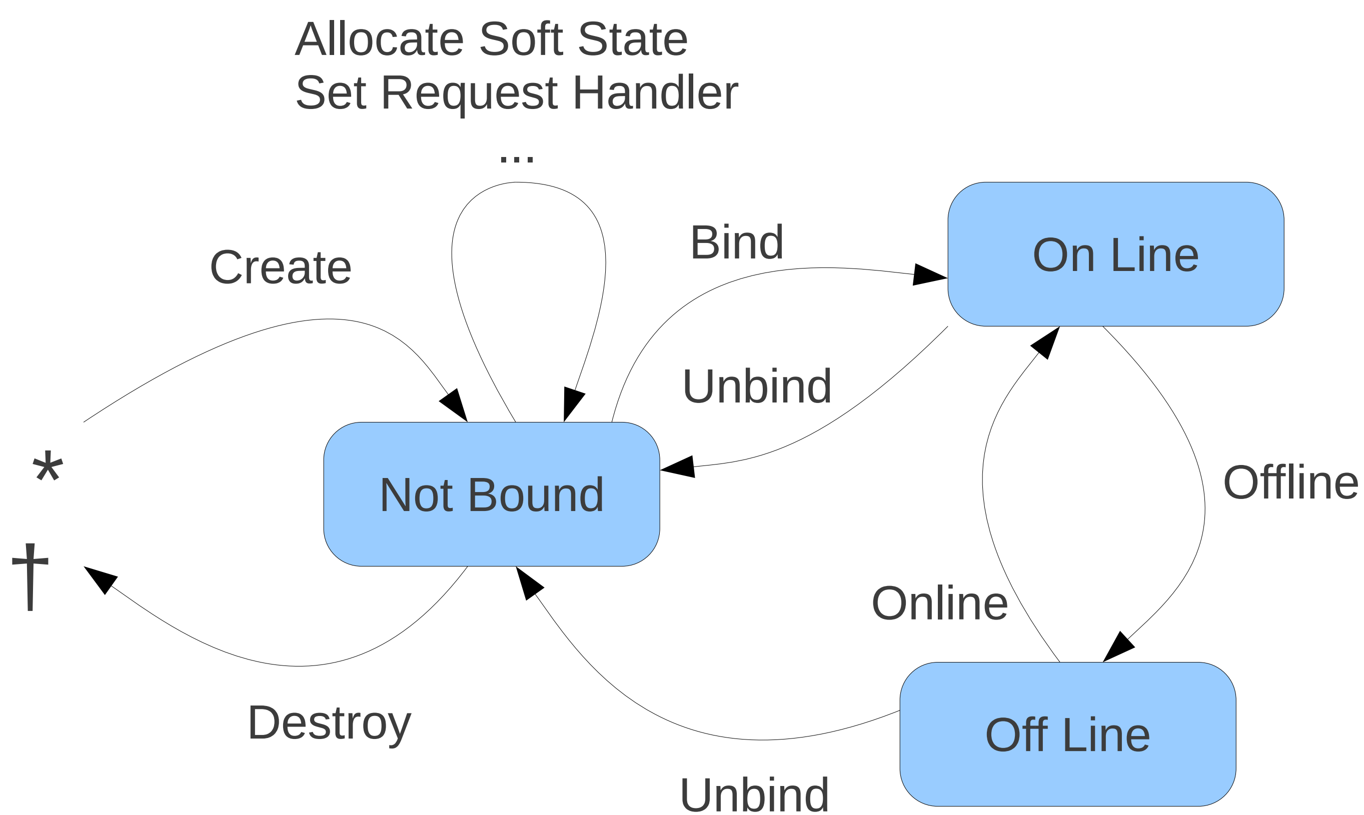
- Transition = driver entry point called



Function Life Cycle

HelenOS

- Transition = driver calls DDF



Hot plug and unplug

- Hot addition – no special support
  - simply later call to dev_add()
- Hot removal
  - surprise removal
    - communication with device is lost
    - dev_remove()
  - administrative removal
    - dev_offline()
    - non-forced – fail when there are clients
    - forced – disconnect clients

Exposing Driver Services

Location Service

- Inspired by CORBA paper
- Any task (server) registers any number of services
- Service must be registered with a unique string name
- Service is assigned a numerical ID (service ID)
- A service can be added to one or more categories

Exposing Driver Services

- Clients find services by name or category
- Can register for notifications when contents of a category change
- Example: Input server listens for and opens any device in category 'kbd'

Exposing Driver Services

HelenOS

- DDF exports a device function as a service via LS (name is based on path in device tree)
- Non-DDF driver exports a service via LS
- Both can implement the same IPC interface

- Client looks for a service implementing an IPC interface
- Client knows nothing about the implementation
- Pseudo-drivers (e.g. file_bd) not in DDF

Q&A

HelenOS

Questions?

HelenOS

http://www.helenos.org/

Thank You!