Großer Beleg

ACPI for L4Env

Lukas Hänel

June 14, 2007

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Christian Helmuth

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den June 14, 2007

Lukas Hänel

Acknowledgements

I like to thank Christian Helmuth for his great support as tutor.

Contents

List of Figures

1 Introduction

Computers comprise many internal devices to serve a variety of tasks. These devices can be assigned to two groups: Devices involved in calculations and devices that provide communication. The latter are also called input–output (I/O) devices. The system architecture describes CPU and memory but only an interface for I/O devices. Operating systems run directly on hardware. Their manufacturers have to develop and maintain code for each architecture and each I/O device they want to support. That is usually a big expense. Another problem with I/O devices is that they are no part of the system architecture. Operating systems need means to discover the devices on a computer. In modern computers, these means are the PCI bus and the ACPI system.

All common operating systems like Linux and Windows place their device drivers into the kernel. In this monolithic kernel approach drivers have access to the complete system, although they only serve a single purpose. Malicious drivers can thereby gain control over the computer. Faulty drivers can reduce the system stability, leading to a system crash.

Microkernel based operating systems avoid this problem by placing device drivers in a restricted environment, the userland. However, to use their devices, drivers need access to the I/O interface. Modern hardware and microkernels provide means to allow these drivers to access only their device. Nevertheless, DROPS [DRO], the operating system developed at the TU Dresden does not use these means. Drivers can access all I/O devices because DROPS does not know of any I/O device. That has to change, because the recently designed Bastei Architecture [FH06] strives to use untrusted drivers to save development expense.

The goal of my work is to integrate device discovery mechanisms to L4Env [L4E], a lower layer of DROPS. With the information on devices I shall design a system that manages access restrictions of drivers. This system must not make decisions on its own, but accept policy statements from a policy manager. The discovery mechanisms to use are ACPI and PCI. Therefore an ACPI driver has to be ported to L4Env.

Organization

This work is organized in five chapters that guide from the basics, over to the design decisions and implementation solutions, to application exploration and an outlook to interesting extensions.

Chapter 2 explains device discovery in general and resource management in current systems. Furthermore it outlines related work in driver–device management.

The third chapter shows an architecture for device and driver management. Then I derive the tasks of the components and their interfaces. Finally I extend my design to restricted systems.

Chapter 4 shows the integration of an ACPI discovery library and the implementation of the device manager. It further shows solutions to include legacy systems.

In the fifth chapter I evaluate use cases and present steps toward ACPI virtualization. Chapter 6 contains an outlook that guides the way to support more hardware features. Finally, a conclusion recapitulates the results of this work.

2 State of the art

In this chapter I present important basics for my work. I first show how devices can be discovered and how ACPI works. Afterward I explain the current resource management in Linux and L4Env. I thereby show the problems with the current L4Env implementation. In the third section I discuss related works.

2.1 Device Discovery

Devices in a computer connect to each other via bus devices. These buses then provide means for the operating system to find out about the devices connected to the bus. The communication channels of devices are abstracted to resources. Resources are regions in the address space that have special semantics. To use devices, the operating system must know their resources and their identity.

Old architectures described the whole computer system, so the standard determined resource regions and types for all devices. When computers supported more and more devices the operating systems needed new means to discover the devices.

2.1.1 Plug and Play

The Plug and Play (PNP) standard [PNP94] defines a hardware–software interface to transfer device descriptions. The standard defined the *EISA ID* as 4-byte string that identifies devices. The first two bytes define the manufacturer, who can define the meaning of the remaining two bytes. Another 4-byte string, the unique id allows PNP systems to differentiate between multiple devices with the same EISA ID.

The PNP standard describes resources of a device as buffer of resource descriptions. Each resource has a type and appropriate properties. The IRQ resource states the IRQ number and whether the IRQ is low or high, level or edge sensitive. For I/O Ports the standard states, whether the device decodes the full 16 bit ISA address or whether it only decodes 10 bit. Furthermore the resource descriptor tells the range in within the resources of that device can be allocated and, which alignment restrictions apply. The closing field states the number of I/O ports requested - the size of the resource. The end of the buffer is indicated by the End Tag resource.

Operating systems can query these descriptions by walking through all devices. For each device the operating system has to read and acknowledge every single byte through one register. Many problems hindered this standard to become a success.

2.1.2 Probing

If an operating system does not get enough information on a computer, it can guess. Instead of loading only the right drivers the operating system can load every driver. If the driver is successful, a device was discovered. That is, of course, a lengthy operation. It is also possible that a wrong driver causes damage to devices. Attempts to access a nonexisting device could lead to system halt. Therefore the probing operation requires much care and should be considered as final option. However, certain guesses seem to be right all the time and programmers consider them save.

2.1.3 PCI Architecture

The powerfull but easy to use interface of the PCI bus benefited its success. PCI devices report their identifier and resource needs in their PCI configuration space. The PCI bus allocates the resources of its devices and provides an easy means for the operating system to obtain that information. It also provides a simple way to discover all devices with identifiers. The first part of the PCI configuration space has to contain the PCI ID and other properties defined in the standard. The remaining part can be used by the device and its driver. After the adoption of the standard, new requirements emerged. Due to drivers already occupied the remaining part of the configuration space, it was not possible to introduce new standard features in this part. That is because the PCI standard addresses properties by configuration space address, not by a name.

2.1.4 ACPI

Although the PCI bus provides ideal device enumeration and configuration, not all devices use the PCI bus. Many devices do not need the high bandwidth that PCI provides. Also the required complexity of the PCI interface is not adequate for many devices. Legacy devices are perfect and computer users will use them in the future. Still, old devices bring along the old buses with bad device configuration possibilities. On the other hand, also new devices can be simple. For example the power button can be programmed to forward the "press" action to the operating system. These devices need a simple discovery and configuration means, too.

ACPI, the Advanced Configuration and Power Interface, was developed as ultimate solution to device enumeration and configuration problems. It shall provide an enumeration mechanism for every hardware. The interface shall be general, allowing to address every possible device and to perform any necessary configuration. Concerning the future, the standard shall support extension by design. A further goal is to push device configuration and power management from the hardware to the operating system.

To be independent from any prior hardware, ACPI is designed as a separate configuration interface. It provides a device database with device descriptions and device interfaces. Devices are ordered in a device tree that reflects the physical hierarchy. For example, the ISA bus stands above all ISA devices. Each device object has predefined fields and methods with names.

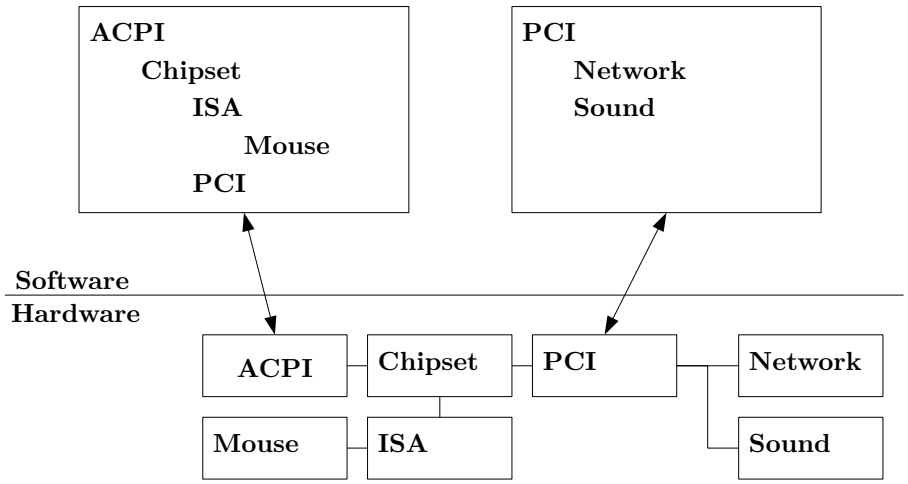

Figure 2.1: cooperation of ACPI and PCI in device discovery

Every device must have an identification field. The bus of the device determines the type of the identifier. ISA devices, or devices with no standard bus use the EISA ID2.1.1. PCI devices are identified by their parent device and the slot address.

All devices also have a resource description. The format of this descriptions is taken from the PNP standard2.1.1. Extensions span additional resource types like memory mapped I/O regions and 64-bit address widths.

ACPI defines control methods to configure a device. These methods can be used to store or retrieve information from the device or to trigger certain actions. For example the *set resource* method allows the operating system to allocate the resources of a device. For every device class ACPI defines suitable configuration interfaces. Power management and other general settings are defined for all devices.

For simple modern devices, ACPI may define all possible user interactions. For example a notebook could use the ambient light sensor to measure the light of the environment. Using this value it could adjust the brightness of the display to save power. The ACPI defined sensor does not have a resource, but ACPI defines one control method to read all possible illuminance levels and one method to read the current illuminance level.

ACPI comprises the ACPI registers, the ACPI BIOS and the ACPI tables. The ACPI tables are static datastructures laying in the main memory on system start up. The operating system has to search for the first table in the BIOS area. This table then contains pointers to further tables. One of these tables states the addresses of the ACPI registers and the number of the ACPI interrupt.

Most important for device discovery is the biggest table, the *DSDT*. The Differentiated System Description Table contains the device tree. *AML*, the ACPI Machine Language is the byte format of the DSDT. Motherboard manufacturers can use ACPI on their motherboards. They can write the DSDT in *ASL*, the ACPI Source Language. That language is written in plain text and allows the definition of devices and control methods. A compiler can be used to generate the AML format from the ASL file.

The operating system uses a parser to load device descriptions and function into a namespace. The operating system can use values from the namespace immediately. An interpreter can execute the control methods. These methods read settings from I/O memory into variables, encode them and return values in the right datatype. The other way round is also possible. Additionally, they can access the PCI configuration space, acquire locks or access the physical memory.

The names of all objects, properties and control methods are bound to 4 letters. The *_HID* field contains the EISA ID. Because most simple devices use standardized protocols, these IDs are not vendor defined. Instead, the Microsoft defined set of compatible IDs is used. These IDs all start with "PNP0" and are defined in a Microsoft document [PNP]. For example "PNP0A03" denotes the PCI Bus.

The ACPI registers are used to transfer control from the hardware to the operating system, to configure power management and to determine the origin of the ACPI interrupt. That interrupt is triggered for example to indicate events from devices, like ejection.

ACPI states the datatypes for all fields and method parameters. However, it does not state, how vendors have to obtain these values from their hardware. That way ACPI acts as an abstraction layer from low-level hardware protocols. The price for this variety of features is paid in complexity of the device description and complexity of the ACPI driver.

2.2 Resource Management

When the operating system knows about the devices and resources of a computer, it wants to assign each device a driver. Most devices are to be driven be one driver at a time only. To prevent interference operating systems provide mechanisms to synchronize driver accesses. That way it usually is impossible that two drivers access the same resource at the same time. To enforce this policy, operating systems have a resource management system. Such a system takes track when drivers reserve resources. Such resources are then only accessable by that driver. Other drivers can only acquire free resources.

2.2.1 Linux

In Linux [CRKH05], resources are managed by datastructures and access functions. The resources are hierarchical ordered by the containment relation. On kernel start up the resource tree is initialized with just one top resource. That resource spans the whole address space. Device-discovering systems like ACPI and PCI split that resource into leaves and subnodes. Whenever such a system discovers a new I/O regions it searches the smallest leaf in the resource tree containing that region and creates a new leaf. Also it links that new leaf to the discovered device. That operation is also permitted to drivers that find it usefull to split up resources of their device. To prevent defective drivers from accessing wrong resources, attempts to get resources that overlap in the resource tree are denied. Nevertheless drivers could just not adhere to the protocol and access I/O regions directly.

2.2.2 L4io

The DROPS project uses drivers from existing operating systems like Linux and FreeBSD. These legacy drivers are stripped from their original system and integrated into DROPS by an emulation library. Each driver is implemented as a separate server. Thus the emulation library cannot synchronize requests of all drivers. Instead the system-wide I/O synchronization is done in a central server, *L4io* [Hel01]. L4io provides resource management, PCI operations and interrupt access.

L4io's resource management is close to the Linux resource management system. Drivers can request resource regions from L4io. If the resource is free, L4io grants the driver the right to access the resource. Afterward no driver can successfull request this resource. When the driver is closing, the driver can release the resource. Thereby L4io enforces a weak policy that two drivers may never use the same resource at the same time.

L4io has two means to discover I/O resources. When it has initialized the PCI library it grabs the resources reported by PCI devices. These resources are stored in the announced memory regions list. L4io has no further means to discover devices on its own. Instead, it relies on drivers requesting resources for their device. Whenever a driver requests a resource not yet managed, L4io notices it. It identifies the resource by type(port or memory space), start and length. Then it registers that resource in its resource management.

L4io provides the PCI operations through an IDL interface. This interface is close to the Linux PCI library. Drivers that want to interact with a device, first have to search for a device handle by the PCI ID. There are interfaces to search by vendor and device id, by PCI class or by the slot. The returned object contains resource information and a handle for further communication. With that handle drivers can invoke all PCI operations in L4io. The current implementation allows drivers to configure every PCI device.

The interrupt implementation in L4io is taken from the *omega0* reference implementation [LH99]. Drivers can attach to all interrupts without restriction.

In [Fri04, 3.4.1 Handhabung unter DROPS (page 23)] Thomas Friebel points out that a new version of L4io should allow drivers to access their device only. In this work I present the design of a new version of L4io.

2.3 Related Works

2.3.1 Microsoft Singularity

The Microsoft Singularity [SRH+06] project develops an operating system that enforces component isolation by a strong type system. Drivers are Software Isolated Processes that have a device driver manifest. This manifest contains a list of resources the driver needs and a list of interfaces the driver will be providing. The manifest acts as a contract between the operating system and the driver. Drivers without manifest cannot be installed. Low level hardware functions are abstracted by a managed environment. Drivers have to use this abstractions to interact with their hardware. As drivers are

written in C#, the operating system can decide, whether drivers use the right resources. Thereto the operating system compares all calls to the hardware abstraction with the resource needs stated in the manifest. If a driver does not comfirm to its manifest it will not be installed. After the driver is installed and before it is run, another check is performed. The system now verifies that the drivers demands do not conflict with the current allocation of resources.

The concept of driver metadata is powerfull and should be adopted to other operating systems as well. The driver verification is, however, only possible due to the new language concept. Thus, their operating system cannot reuse legacy drivers, rendering this approach unusable for a small research team.

2.3.2 Apple I/O Kit

In their current operating system, Mac OS X, Apple introduced the I/O Kit [App06]. The I/O Kit is a comprehensive approach to driver development and management. It comprises a framework and libraries that accelerate driver development and result in robust and interoperable drivers. Alongside with enhanced driver writing comes the I/O Registry. It is a managing component that connects devices to drivers, drivers to their family class driver and family class drivers to applications. The framework contains implementations for all family class drivers and provides clear and unified interfaces between all components.

The I/O Registry uses a platform expert to find initial devices. Each component provides and requires its services via so called nubs. First, the I/O Registry registers nubs provided by the platform expert. It then searches through the I/O Catalog, a driver database, to find possible clients for that nub. Afterward it initiates the probe command for each returned driver. The driver with the best score is chosen to drive the device. After initialization, the driver provides nubs as well, which the I/O Registry will process further. Example nubs provider are buses, providing nubs for each connected device and family class drivers, providing application interfaces.

The I/O Kit has a remarkable driver management architecture. However, drivers are not forced to comply to the specification. Their are no language based checks verifying that the driver complies to the nub interface. Also, drivers lie in the kernel's address space and can degrade system security and stability.

3 Design

The current implementation, L4io, does not know about all devices of which the system consists. Therefore it cannot know, which resources will be requested by drivers. Missing that knowledge it has to trust all drivers to behave correctly and to reserve only resources they need. In contrast the Bastei framework wants to use untrusted drivers, so we cannot rely on drivers to choose what resources they will get.

Instead, a trusted instance should decide what resources a driver may receive. To satisfy our primary goal, the separation of drivers, they shall only receive resources of their device. The trusted instance therefore has to know, which resources belong to a device and, which driver may drive a device.

First, the trusted instance has to get a description of the actual hardware, the set of devices and their resources. This information is available from PCI, ACPI, architectural information, or can be obtained by probing.

The second part is to decide, which driver may drive a given device. Therefore the trusted instance has to search suitable drivers in a driver database. From the set of results the trusted instance then has to choose one driver to be started. It can base that decision on a policy. Such a policy could for example prefer open source drivers over closed source drivers. Other criteria could be the known stability of the driver or the number of supported features. If their are no runtime properties of the drivers known at that time, the trusted instance could try out all drivers. First allowing one driver after the other to use that device, then measuring the properties and afterward unloading the driver.

First I will show the architecture of such a system and then I will detail its interfaces. The remaining part shows how such a system can be applied to systems without ACPI.

3.1 Architecture

The trusted instance wants to restrict drivers from accessing I/O resources not belonging to their device. Therefore the trusted instance has to know, which resources belongs to a device. This instance needs a connection between resource and device. This connection can be established in two ways: Either each resource additional has a device or each device has resources. I chose the latter solution, because devices structure resources and provide a good abstraction. Device description can be gathered from PCI and ACPI subsystems.

Drivers then shall be allowed to use their device. Meaning, they are allowed to use its resources, but nothing more. L4io was already providing exclusive resource usage, helping to synchronize drivers. I designed the *ioguard* as successor of L4io that operates on devices rather than resources. It gathers devices and enables drivers to use them.

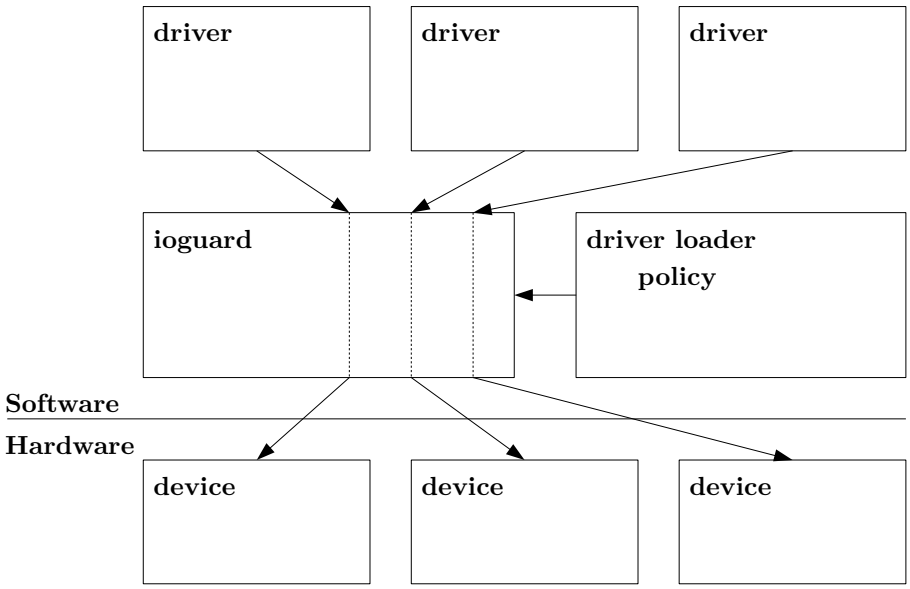

Figure 3.1: Architecture

The management of drivers, as described previous was not the task of my work. Instead, I provide an interface to such a system. This system I call *driver loader*, because it shall select drivers and start them. It can therefore rely on the ioguard's devices and tell the ioguard for each device, which driver can access it.

To use the ACPI and PCI system and to be able to grant access to I/O resources, the ioguard interacts with the hardware and owns most hardware resources. The driver loader instead does not need any hardware access, but may be integrated into upper parts of the system, like a filesystem or network connection. In my work I show that a minimalistic interface between these components suffices to get them work nicely. Thus I have implemented them in different servers, dividing enforcement(ioguard) from policy(driver loader).

In common systems, drivers search through global data to get device information. In L4io, this approach was moved into a client-server scenario, where the data and the search is on the server side, and client requests are transferred as well as the results. Nevertheless the driver could search for and use all kinds of devices. In my design I inverted the schema: The device the driver shall drive is already determined by the decision of the driver loader. Therefore I now need a means for the driver to get to know, which device it shall drive. That information could be passed from either the driver loader or the ioguard. I do not develop the driver loader to driver interface here, so I would not like to pass this information that way. The ioguard already has to distinguish between drivers, and knows, which device each of them shall drive. So drivers can just ask the ioguard about the device they are supposed to drive.

3.1.1 Ioguard

The ioguard maintains a device tree and implements means to build it up and to operate on it. It inherits L4io's position in the system as center for I/O resources and I/O operations. In addition it is part of a device and driver management. It maintains semantic information about devices whereby it strengthens its position against drivers.

It does so by managing devices and their resources. It knows in advance the I/O resources a drivers will use. It takes devices and their resources from ACPI and PCI systems.

To take device descriptions from ACPI I needed an ACPI software implementation. There are two main implementations, one from Microsoft and one from Intel. The Microsoft implementation is not public, preventing me from using it. I therefore used the Intel implementation. Their library, the ACPI component architecture (ACPICA)[ACP03], is an operating system independent free software, also used inside the Linux and the BSD kernel. I choose the pure version from Intel because the latter two are adapted and highly integrated into their environment. Using the ACPICA library, I am now able to extract devices and their resources from the ACPI system. Often the devices physical hierarchy is reflected in the ACPI tables. I use this hierarchy to build up the device tree.

The L4io PCI library strips the resources of all PCI devices and reports them to the resource management. I modified this to receive devices together with their resources. I place all PCI devices beneath their bus device in the device tree.

After all subsystems are initialized and the device tree is complete, the ioguard waits for policies from the driver loader and for requests from drivers. These interfaces are described in the following sections.

3.1.2 Driver Loader

To start drivers for each device, the driver loader needs to know all devices. It gets this information by requesting the device descriptions one after the other from the ioguard.

Afterward the driver loader selects appropriate drivers for each device. It then tells the ioguard, which devices each driver may access. Therefore driver loader and ioguard must agree on identifiers for devices and drivers. The ioguard assigns each device an id and transfers the id in the device description. To refer to a driver I currently use the task id of the driver. Using this ids the driver loader creates a capability in the ioguard. Each capability allows one driver to use one device.

When all rights for the drivers are set up, the ioguard loads and starts the drivers. If a policy changes and the driver loader has to assign a device to another driver, the driver loader wants to change the capabilities. Therefore it can destroy the capability of one driver and create a new capability for another driver.

There exists another idea to transfer policies from the driver loader to the ioguard. Now I want to discuss this approach and the reason, why I did not chose it. In my design the driver loader asks the ioguard for the device list. Then it tells the ioguard the drivers for each device. In the complementary idea the ioguard asks the driver loader about the driver for each device. The driver loader does not push policies into the ioguard,

but the ioguard pulls policies from the driver loader. When the ioguard requests the driver for a device, the driver loader needs a description of the device. Therefore that descriptions has to be part of the request. The driver loader could then decide for a driver and return a driver identifier. Later I will discuss the requirement that one device may be driven by several drivers. In that case the driver loader answers with a set of drivers, thereby introducing complexity.

The new approach is equally to my design, if the driver loader can always decide for the right driver. That might be possible if the driver loader has a configuration for the current hardware. If not, the driver loader has no complete view on the computer and might not be able to make decisions. When the driver loader needs to know about all devices before it can decide for a single device, an additional interface is needed. The driver loader first has to setup default capabilities and change its decisions later. An additional interface should tell the ioguard of a change in the decisions of the driver loader. A simple notification would implicate that the ioguard has to walk through the complete device list again. An improved version would allow the driver loader to specify the device too. The same interface is needed when policies change and the driver loader has to alter its decision. It is clear that the driver loader needs a possibility to push new decisions into the ioguard. I posed myself the question, why this interface should not be used in the first place. I therefore decided to use my design, it separates data flow (export of device description) from control flow (capability creation).

3.1.3 Driver Interface

As motivated in the architecture chapter a driver has to ask the ioguard about the device it is supposed to drive. In the usual case one task contains one driver that drives one device. Another possibility is that a driver is implemented as thread or function in a library that is part of an application. Furthermore that application could also use several drivers for several devices. In that case one task would need access to multiple devices.

Therefore all drivers have to ask the ioguard for a set of devices. They thereby get a description and resource list for each device. To access resources of that device, drivers have to ask the ioguard. Therefore drivers and the ioguard have to agree on a device identifier. The ioguard derives this identifier from the capability of the driver and embeds it in the device description. For the driver that id appears to be the device id. When requesting a resource, drivers specify the device id and the appropriate resource. With that information the ioguard checks, whether the driver is allowed to use that device. If so, the resource will be made accessible for the driver.

To use the PCI operations provided by L4io, drivers first had to search for their device and then they could call configuration functions. Thereby drivers could also gain knowledge about other PCI devices and could also call PCI operations on them. Instead of searching for a PCI device handle, drivers now can use the ioguard device id. That way they do not get to know about other PCI devices. Thus I removed the L4io PCI device search functions, but I kept the PCI configuration interface. To read their devices PCI configuration space, to enable busmastering for their device and to set a power mode, drivers now specify the ioguard device id instead of the PCI id.

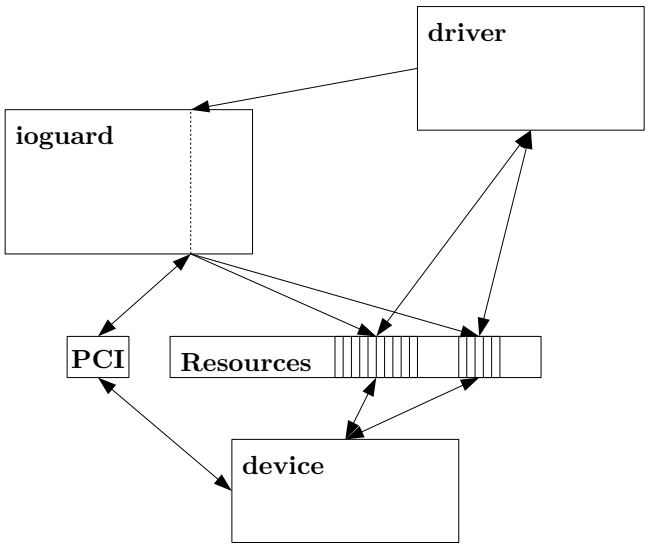

Figure 3.2: Driver Interface: The ioguard transfers resource rights and provides PCI operations

A similar interface can easily be build for the ACPI library. Drivers would also use the ioguard device id and invoke configuration functions in the ACPI system. As pointed out in the foundation chapter 2 ACPI supports configuration functions for many device types. To give a small example one possible interface is the brightness control of a laptop display. ACPI defines functions to query the current brightness level, to query the list of all levels and a function to set the brightness. This interface could be exported from ACPICA and a display driver could use it. The focus of this work was device management and so I did not implement ACPI defined configuration interfaces. However, in the outlook chapter 6, I outline further problems that will arise.

3.2 Details

3.2.1 Device

To this point I talked about devices as highlevel units, whereas drivers need to know about the device's communication channels. Also the driver loader should be able to distinguish devices and find drivers matching to a given device description. I therefore derived a device object consisting of a description and a collection of resources.

The driver loader and drivers can express a device reference by the ioguard device id. However if they get a device object from the ioguard they have to know the type and name of the device. The best available description is that one given by the device discovery system, in my design the ACPI or the PCI system. So an ACPI device is identified by the triple EISA ID, unique id and compatible EISA ID. For PCI devices I chose the common tuple of PCI class, vendor, device, sub_vendor, sub_device, bus, devfn. The properties bus, devfn for PCI and unique id for ACPI are the devices addresses and therefore no device types. They do not serve the purpose of finding an

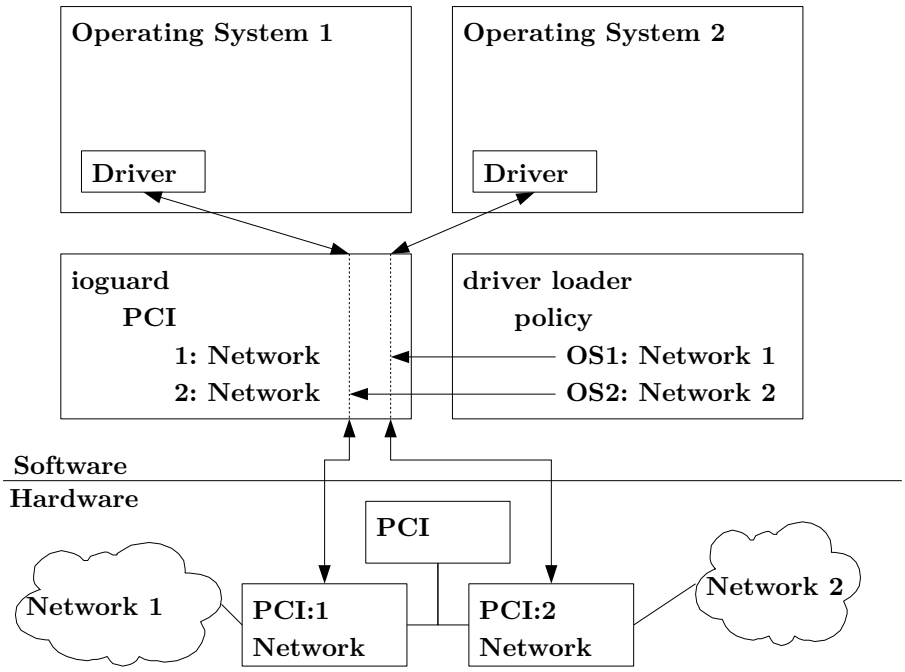

Figure 3.3: The address of a device can be important for a policy

appropriate driver, but to identify the instance of the physical device. Therefore they might be important to a policy that wants to assign a specific driver to a specific device. Consider, for example a server that runs several virtual machine and is connected to several networks through different network cards. This server could be interested to grant each virtual machine access to only one network. Therefore it would grant the virtual machines access to an appropriate network card. Drivers that handle several devices could be interested in a persistent way to distinguish devices, too.

On current x86 architectures I have three important input–output resources: Interrupts to signal asynchronous events, I/O-ports to access registers and minor buffers, and I/O-memory-regions for page-aligned buffers and registers. These resources have different hardware interface and different meanings for their position and size. So the ioguard has to handle them each in a different way and store different properties for them. Therefore I store them in different sets. The device object that drivers and the driver loader get from the ioguard contains three arrays, one for each resource type. Drivers can then inform themself about the number of resources for a given type. To use a specific resource, drivers simply tell ioguard the device the resource belongs to, the type of the resource, and the number of this resource in the resource list. The ioguard will then transfer the right to use that resource to the driver.

3.2.2 Capabilities for fine granular device access

For now I assumed that a driver gets full access to all the resources of a device. A broader approach is useful in the scenario that a device may contain several subfunctions that

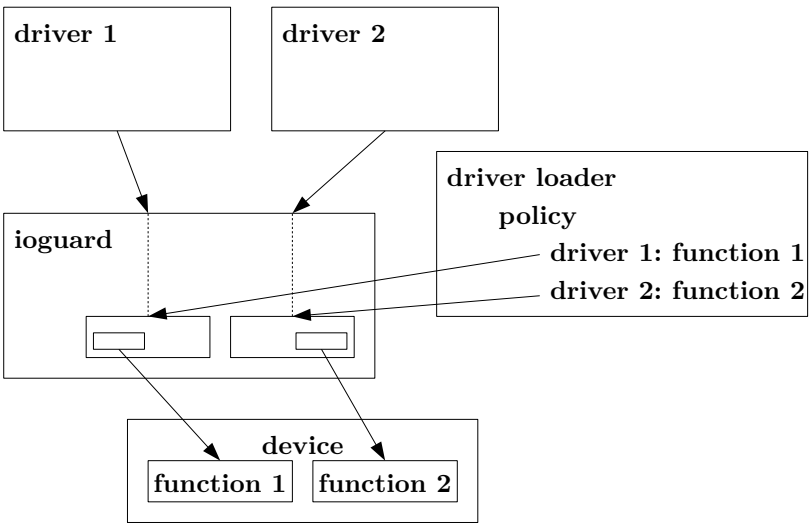

Figure 3.4: Partition of a device between two drivers.

do not necessarily have to be controlled by one driver. For example a network card with two jacks connected to different networks, might be implemented as two distinct logical devices that only share the same board. They might appear in the system description as one device that has two complete sets of network card resources. A policy could then like to split the reported device into two parts and grant different drivers different parts.

Giving the example there might be an additional I/O resource that would allow the control of both network cards. One possible feature could be to bridge the two cards such that they appear to be in the same network. Such an interface would be controlled by one driver, but it might be useful to the other driver to know about the current bridge setting. In that case a complete partition would not be satisfiable, instead two drivers should be allowed to use the same resource. A further desirable option in that example would enable the driver loader to grant one driver read-only access to that resource and full access to the other driver. If a device can be controlled by different drivers simultaneously, or if two drivers synchronize their device access on their own, to give both of them full device access might work, too.

One of a devices subfunctions could provide an auxiliary feature, and a driver, although claiming to support these feature, causes this feature to malfunction. So even if there is only one driver for a device, it might be preferable to deny it access to certain I/O regions.

I therefore developed an access-control list per driver-device pair stating for each of the devices resources, whether the driver is allowed to access it. I call this list, together with the driver-device pair, a capability. This capability shall be derived from a policy in the driver loader and transferred to the ioguard. The per-resource access rights can be modified by the driver loader in case of policy change and are destroyed together with the capability, when the driver loader revokes the right of the driver to access the device.

With the capability configuration a driver loader can grant a driver a subset of the devices resources. If the driver loader wants to split up a device it grants two drivers access to disjoint subsets of the devices resources.

In the same manner all interfaces the ioguard provides can be restricted. For each PCI operation one could add a right to the set of capability and check it whenever a driver tries to call this operation. As for the PCI config space one could also create logical regions, for example one for the area defined in the PCI standard and one for device specific areas. After adding them to the set capability, driver loaders could deny drivers to write to the standard-defined areas.

3.2.3 Dataspaces

The rights to access an I/O region are transferred between address spaces by the means of the L4 mapping IPC. This method is also used in L4io to transfer I/O regions. Although abstracted by the IDL interface it is still close to the kernel and the user land is not informed about the new semantics of the modified part of the virtual address space. The user land - except the driver of course that will expect the region to be accessible until a release call. But, in a dynamic environment that is not guaranteed. It is possible that the I/O region is unmapped during the runtime of the driver. So when the driver accesses it, the kernel will send a pagefault to the pager of the driver. L4env uses the concept of dataspaces [APJ+01], so the region manager L4rm is the pager of the task. The region manager now shall forward the pagefault to the right pager for the affected area. As its region map was not affected by the kernel-level map operation it cannot give an answer in that case.

Dataspaces were introduced as a means to overcome the problem of uninformed pagers. Unstructured data containers called dataspaces are provided by dataspace managers. Applications can establish access to a dataspace by telling their region manager to attach the dataspace to the virtual address space. Thereby the region manager will also insert the dataspace in its region map. A dataspace is represented by the task id of the dataspace manager and a dataspace id provided by the manager. So in case a mapping is removed afterward the region manager can remap the affected page from the dataspace manager.

Appling this concept to my design, the ioguard becomes a dataspace manager providing dataspaces on top of I/O regions of port and memory space. Several dataspaces, each providing an IDL interface, are already implemented in L4Env. The question of reuse arises. The most general interface is the *dm_generic* interface that implements most dataspace operations. The identify and create calls are defined in the specialized interfaces. For example, the next interface in the row, *dm_mem* handles anonymous page-aligned memory. In contrast I/O regions are predefined and have a special meaning. The dm_mem interface is therefore not applicable. The other dataspace interfaces are build upon dm_mem, so they cannot be used either so I only evaluate dm_generic.

Dm_generic has no means to tell a client about available dataspaces and does not state how dataspaces are created. An extension is therefore required. At first I have to decide who creates dataspaces. Either the ioguard has to create dataspaces on top of each resource or drivers can create dataspaces. I did not choose the first approach,

because the ioguard cannot know in advance how many dataspaces a driver will need. The second approach includes the design of a create dataspace call that returns the dataspace id of the new dataspace. Thus the driver knows about all dataspaces on its device.

To create a dataspace drivers address a resource by means of device id, resource type and resource number. Additionally they indicate the desired position and size of the dataspace. To finally access the resource, drivers instruct their region manager to attach the dataspace to the virtual memory.

Now that dataspaces can be created and closed, there was no need for the L4io request and release memory or port region functions. Consequently they have been removed from the io interface.

3.3 Compatibility - Portability

My system is based on information from PCI and ACPI hardware, but what happens on systems that miss them? I want to support legacy systems that do not have ACPI support, or even no PCI-bus. Embedded systems with neither PCI nor ACPI shall be supported, too.

Systems without ACPI-support will not tell the operating system of all devices of which they consist. This additional information is usually encoded in drivers. If I want to use this hardware, I would have to fallback to the schema, where I do not know what drivers will do, and where I allow them everything. I could then either use an uninformed mode in my software, or just reuse the old software. Either way I would have failed in separating the I/O space. I want to apply my schema that involves the ioguard. As the central instance it has to know about all devices, so I need to provide it another interface to discover devices. The simplest way is a static file that describes the system configuration. The user has to provide such a file that I call the *static configuration*, at boot time.

A static configuration, describing the systems device configuration has to provide device objects with device identifiers and resource lists. A device identifier should use an existing device identification mechanism, so both drivers and the driver loader would not need much adjusting. They already know the EISA ID, so that seems to be the best available device identifier. The PCI ID is not appropriate because PCI devices need a PCI bus that would then solve the discovery problem.

If I attempt to initialize the PCI bus on systems where it is not found, I might produce errors. This error occurs, because the PCI bus driver probes for its device. On such systems I neither need this driver, nor the PCI library. To avoid probing for the bus, I have to get information about its presence from another source. On systems that with ACPI support, I can inspect the ACPI tables, but more general, I have to rely on a static configuration. To avoid probing in all cases I assumed that either way my program is informed of the presence of a PCI bus. Only if either ACPI or a static configuration informs the ioguard about a PCI bus, the ioguard will initialize the PCI library.

3.3.1 Probing

I thought of several ways to obtain a static configuration. Before the rise of discovery techniques, the device configuration was given in the architecture specification. So I could derive a static configuration from the architecture. I would do this for all architectures I like to support and ship these configurations with my software. But, as stated in 2, a static configuration might not be sufficient and probing is the only way to obtain more information. There are several places, where I could do the probing.

First, I could allow drivers to probe, meaning, I allow probing on all possible places of the device and ask for the result. I then revoke the places where no devices were found. Nevertheless, I have to allow drivers to access the whole I/O-space, and trust them to claim only the resources they need.

Second, I could extract the probing code from all drivers and do it in the ioguard. Given, probing may lead to delay or even blocking, I cannot guarantee that my service is always available. Furthermore that means a rather high programming and maintenance expense. That is because it might be difficult to separate probing from running code and this separation is likely to change on each code reorganization of the driver.

Thirdly, I could run an untrusted operating system on my hardware. After it has probed and found all devices I use a tool that extracts the device configuration from the running system. This operating system would gain full access to all devices of my computer. In contrast to the first approach this operating system is then also able to use all these devices, reading any data and using network connections. Of course I could unplug all network connection, to prevent the system from leaking unprotected data. A malicious system could nevertheless override the harddisk and try to burn the CPU, for example by deactivating emergency shutdown, halting the cooler and doing intensive calculations.

Neither of this possibilities was designed here, I just propose to use a static configuration on such systems. The minimal information the configuration file has to provide are the devices of the system. Each device should be described with an identifier and a resource list.

4 Implementation

I choose to explain three challenges in my design. There were three main problems: The integration of ACPICA, the layout of the internal data structures and the integration of dataspaces. Furthermore I present an intermediate solution for a static configuration and the integration of my design into existing drivers.

4.1 ACPICA

The source files of ACPICA are subdivided into folders that refer to components that are called subsystems. Each component provides a certain interface but uses possibly all other components interfaces. There are two interfaces to the operating system, one demanding basic operating system functionality and the other providing the ACPI services. You integrate ACPICA by implementing the lower interface described in the file acpios.h, and invoke functions from the acpixf.h header file.

The operating system interfaces required by ACPICA were: synchronization primitives, anonymous memory allocation, PCI operations, I/O port access, physical memory access. They were mapped to the appropriate L4env services. ACPI AML code can invoke PCI operations, for example to read additional properties from the PCI configuration space. I mapped them to the internal PCI library of ioguard thus introducing a dependency between these components. Meaning, I have to initialize the PCI library before I can execute such AML code. Fortunately such code is not called during the initialization of ACPICA.

Starting the ACPI system with ACPICA involves several steps. After each step a wider functionality is provided by ACPICA. Thus the operating system can place these steps at positions in the start up sequence where it needs the appropriate results. For my design only the end result is needed, so I can start ACPICA at a whole. The steps included in the initialization are explained now. First ACPICA searches and loads the ACPI tables, thereby parsing them and filling up the namespace with devices and definitions. After that ACPICA requests access to all I/O regions needed when executing AML code, for example to talk a proprietary protocol through an ACPI defined interface. In the next step the control is moved from the ACPI hardware to the operating system. Therefore ACPICA installs an interrupt and writes an appropriate value to an ACPI defined register. After that ACPICA executes ACPI defined device initializations, possibly also to tell each device that the operating system is in control now. Therefore ACPICA walks through all ACPI enumerated devices and calls their status() (_STA) and, if defined, initialize() (_INI) functions.

To interact with the ACPICA library, I used a subset of the upper ACPICA interface. Besides a call to start the ACPI system and to end it I needed interfaces to access device information. Therefore I first needed a mean to find the system bus, the root

of the device tree. I exported a function that takes an EISA ID and returns a device handle. Calling this function with ¨_SB_¨ returns the first device handle. With a tree walk function, the ioguard gets all further device handles. The next function retrieves a device description from the device handle. This description contains common properties of all devices, mainly the EISA ID, the unique id, and a name. The data type of this description is exported from the ACPICA type definition, so I can just recast the pointer. That way I saved one copy operation in my layered design. Nevertheless, the ACPI subsystem of the device manager (acpi.cc) has to adopt to changes in upcoming ACPICA versions.

The same approach is chosen for resources. The PNP resource buffer exported by a walk call, too. There are 17 resource types defined by ACPI, but I only exported ports and IRQs. I omitted memory mapped I/O regions, because I never encountered them on any ACPI defined devices at three testmachines. The other resources, like address spaces, DMA channels or fixed memory regions are not yet consumed by anyone. Therefore the device manager does not register them.

4.2 Device Manager

The device manager is the part of the ioguard that manages devices. It is written in C++ using a reduced C++ implementation and a minimal library (list, logging, AVL tree). The device manager holds the device tree and a client list. It initializes the ACPI and PCI libraries and implements the io protocol. Devices are classes with resource lists and description objects. The device tree is implemented by children lists for each device. The description objects for PCI and ACPI implement methods to build up their device trees.

In the start up process, the device manager asks the ACPICA library for a handle to the PCI bus device. If that handle exists, the PCI bus is assumed present and the PCI library is started. That handle is also saved in the device manager. Afterward the device manager walks over all PCI devices and inserts them as children to the PCI bus. Then the device manager walks over ACPI devices and inserts them into the device tree, too. If ACPI provides additional information for PCI devices, the device manager will attach these information to the appropriate devices.

The device manager then waits for the driver loader to creates capabilities. Each capability call contains the taskid of the owner of the capability. The device manager maintains a list of all owners in the client list. Each client has a capability list, where all its capabilities are stored. When a driver requests its devices, the device manager first iterates the client list to find a client with the taskid of the driver. If the driver is not listed, the request will be rejected. Otherwise the driver can request device information for each capability. Within the device information the driver receives a device id for further communication. This id is the pointer of the capability object. When the driver requests a device, the capability list is seached for this pointer. Only when the capability is found, the driver manager allows the requested operation.

Each capability contains a pointer to the client and to the device object. Furthermore capabilities contain lists of all resources of the device. The elements of these lists are

called resource capabilities and state, whether the driver is allowed to use the appropriate resource. If the driver tries to access a resource, the device manager also checks these resource capabilities.

The interrupt implementation is taken over from L4io. I only added admission control to the attach call. As the omega0 protocol does not state the capability or device id, the usual resource checks are not possible. Only the taskid of the driver and the interrupt number is provided. Therefore the device manager checks if the client is registered and whether one of its devices uses the requested interrupt. If so, the request is granted, otherwise rejected.

Drivers use the capability pointer as device id for PCI operations, too. The device manager needs a handle to access device objects in the the PCI library. It takes this handle from the PCI description of the device addressed by the capability.

4.3 Ioguard Interface

driver loader

To receive the device list the driverloader iterates over the device tree by calling get_-next_device. It thereby receives a data object describing a device. This description contains resource lists and an EISA ID or a PCI ID to identify the device and an ioguard device id. The driver loader uses this unique ioguard id to reference this device in further communication. It will first use it as reference parameter to proceed with the next get_next_device call. To create a capability the driver loader tells the device manager the reference id of the device and the taskid of the driver. It therefore uses the create_capability call. To revoke that capability it uses destroy_capability with the same parameters. Fine granular resource restrictions can be transferred using a data object that states the rights on each resource. These rights can be altered using the modify_capability call.

driver

The driver interface to request devices is almost the same like the driver loader's get_-next_device call. It also receives device identification objects, resource-lists, and the ioguard id. Resource types are memory-mapped I/O regions, Port I/O regions or interrupts. The driver informs itself about the device's resources, by looking at the resource lists. Each resource gets an implicit number that corresponds to the slot in the resource list. Drivers use the ioguard id, the resource type and the resource number to request the specified resource. To map Port I/O or memory-mapped I/O regions, drivers create dataspace and map them.

4.4 Dataspaces

Dataspaces need a dataspace id to identify the dataspace by client and manager. A dataspace manager unique id would be an easy way to implement it. A natural identification of dataspaces is also possible. Dataspaces are created by a driver using a

capability and specifying a resource. The returned id could only distinguish dataspaces of the same resource. So the global dataspace id would then include the driver id, the driver's capability id and the resource type and number as well as the dataspace number. Such an id could allow the reidentification of the resource, the device and the driver of a dataspace. For the dataspace protocol such an id must be mapped to an integer, thus introducing a conversion step. This overhead might be usefull sometime, but I did not see any benefit. Therefore I use an ioguard global dataspace id.

Dataspaces are implemented as classes with an interface similar to the dm_phys dataspace objects. I mapped the dm_generic IDL calls to the dataspace class.

Dataspaces belong to resources. Also they belong to drivers. They could not have been created without the driver possessing the capability to use that resource. If that capability is destroyed the dataspaces should be destroyed, too. Therefore I store the dataspace objects in a list per resource capability. If the driver loader revokes the capability, the ioguard destroys all dataspaces in the list of that resource.

To access dataspaces by the dataspace protocol, the global id is used. Therefore the dataspaces are also stored in an AVL-tree. Naturally I used the dataspace ids as keys of the tree.

Memory mapped I/O dataspaces can be treated like normal page aligned memory. I/O ports are transferred via an io flexpage, so they are also mapped. That posses no problems to the dm_generic IDL interface. But the dm_generic library is bound to memory pages. It aligns positions and lengths to page boundaries. The I/O pages are however not 4k aligned, so the dm_generic library cannot be used to map or fault these pages. A more general interface might be needed. For reason of time constraints I did not implement such an interface. Instead I just provide a means in the ioguard library for drivers to map an I/O dataspace.

4.5 DDE-Kit

To support legacy drivers from existing operating systems, the device driver environment (DDE)[Hel01] was developed. It is a library that emulates the drivers usual kernel on top of L4Env. The current version divides this task in two parts, the ddekit and the operating system specific part [Fri04]. The ddekit provides a stable interface to L4Env. The operating system specific parts build on top of ddekit and provide the usual kernel interface to their drivers. Currently supported operating system parts are dde_linux26 and dde_freebsd. The ddekit's interfaces are derived from the needs of this operating systems. Especially for resources, ddekit provides the usual Linux request_region, request_mem_region and release calls. In the former L4io version these calls where directly mapped to the l4io_request counterparts. As in my design there are no more request_region functions, I build glue code to map the old interface to the new ioguard primitives. That way I show, how legacy software can still use the new interface.

In my design the operating system already knows, which resource a driver shall use. The driver has to inform itself about the location of the resource and has to create dataspaces to use resources. These tasks are now done in the ddekit. When the driver is started, the ddekit retrieves the list of devices from the ioguard. If a driver requests

a resource by position and size, the ddekit walks through the device list for a suitable resource. It then creates a dataspace with the desired size and attaches it. The ddekit then returns the virtual memory address of the attached dataspace to the driver. If the driver releases this resource, the ddekit wants to detach and close the dataspace. The driver gives the ddekit a virtual address to release. Therefore the ddekit needs to find the appropriate dataspace. As the region manager already maintains a list of dataspaces, I use l4rm_lookup to get the dataspace id back. As this look up is only provided for the memory space, I implemented a small list for I/O dataspaces. This list is filled on dataspace creation and searched through on resource release. Dataspaces found are then closed and removed from the list.

To support PCI devices, the ddekit implements a virtual PCI bus. This bus is implemented as a list of device objects. This list is assembled by retrieving all PCI devices from L4io. All PCI functions that search through the devices or that read properties from device objects work on the local PCI bus. Only operations that interact with the hardware use L4io. This virtual PCI bus could be adapted to my design easily. Instead of filling the virtual bus with all devices, the ddekit only retrieves the devices the driver is allowed to drive. Thus the list of PCI devices contains only information that is accessible to the driver.

4.6 Static Configuration

The way ACPI perceives the device tree does not involve writing or probing I/O memory. Also the DSDT is static and cannot be changed during runtime. So a DSDT can be seen as a static configuration provided by the hardware. With ACPICA it is possible to exchange the hardware tables with tables from the user. My idea was to use ACPICA to interpret user provided DSDTs as static configuration on systems without ACPI. It turned out that it is possible.

Using a DSDT as static configuration file surely is expensive in terms of the required library. However, as the library is already integrated, it was an easy step to get a preliminary implementation. Also, the DSDT is possibly the most powerful device description. Because most fields in a DSDT are optional, a suitable table is possible. This table then only contains device identifiers and resource lists.

If I use ACPICA for static configuration, I need the namespace subsystem. I use it to parse the tables and act as device database. The question is, whether the namespace subsystem needs further subsystems, and what influence they might have. Fortunately, no other subsystems are needed. Thus, the ACPICA is not able to execute control methods or to access hardware. Therefore no side effects occur while using ACPICA for static configuration.

The possibility to exchange the DSDT was introduced to compensate mistakes in the hardware DSDT. If such a configuration contains discrepancies from the specification, the user cannot change it. Therefore the ACPICA library provides an interface to accept a DSDT from the user. Users can export their hardware DSDT, correct mistakes and then configure ACPICA to use the corrected DSDT. I supported that feature, too. Users

```
/* Serial Ports */
Device (UAR1)
{
    Name (_HID, EisaId ("PNP0501"))
    Name (_UID, 0x01)
    Method (_STA, 0, NotSerialized)
    {
        Return (0x0F)
    }
    Method (_CRS, 0, NotSerialized)
    {
        Name (BUF0, ResourceTemplate ()
        {
            IO (Decode16, 0x03F8, 0x03F8, 0x00, 0x08)
            IRQNoFlags () {4}
        })
        Return (BUF0)
    }
}
```

Figure 4.1: Description of the universal asynchronous receiver–transmitter, extract from a static DSDT

can append a DSDT to the ioguard via the bootloader. The ACPICA implementation will then use that DSDT as replacement.

The same interface is used to provide a static DSDT to ACPICA. However, on systems without ACPI, the initialization of ACPICA will fail. I use this failure has indication that a static configuration has to be used. If the user provided a DSDT, my ACPI library will initialize the namespace subsystem and parse the provided table. Afterward the device database is functional. The ioguard can request devices without noticing a change. That is because the current implementation of the ioguard only interacts with the namespace component of ACPICA. As the DSDT should be simple and contain no control methods with hardware accesses the other ACPICA subsystems are not called.

If neither the hardware or software provided a DSDT, the ioguard has no static configuration. To support systems with no ACPI I introduced an ioguard provided DSDT. This minimal DSDT only contains the standard I/O devices from the x86 architecture. It mainly describes the PCI bus and the PS/2 mouse and keyboard device. Thus, if the user forgot to specify a DSDT the PCI system will still work.

5 Evaluation and Virtualization

The ioguard server is used mainly for configuration of I/O devices and the driver system. Once the system is booted and all drivers are set up, the ioguard is only invoked for device reconfiguration or policy changes. After all resources have been spread to the appropriate drivers, the drivers have direct access to their device. Thus, the ioguard has no influence on the runtime performance of the system. Instead it shall ease the configuration of the driver encapsulation. Thereby the ioguard enhances the overall security and fault tolerance of the system. To demonstrate its use and advantages I like to present two use cases, in which the ioguard overcomes previous approaches and shows suitability for future application.

5.1 A modern operating system

I want to show here a possible system configuration and how that fits with my design. This system consists of several devices with an interesting driver constellation.

One part of the system is a a network server that provides network access to the operating system. It therefore uses all available network interfaces and states the current maximum connection speed to applications. The computer shall have a Gigabit Ethernet card, a Fast Ethernet card and a wireless LAN card. The network server implements the drivers for all this devices. To support this network server, the policy of the driver loader grants the network server access to all network devices. On system start up the driver loader will get the device list and create for each network class device a capability for the network server. Having the capabilities, the network server is enabled to use the network cards.

A new graphic system is used in this system, so applications using a graphical user interface (GUI) no longer get parts of the framebuffer. Instead they get their own off-screen buffer, where they can draw their interface. This buffer is then blitted to the framebuffer to display the GUI at the right positions. When the application changes its view, the new content automatically appears on the screen. Window movements and overlapping are handled by the windows management system. The visible parts of the off-screen buffers are then placed at the right positions without interaction with the application. These off-screen buffer reside in memory on the graphic card. Meaning, they are in protected I/O space. In this example, I use the driver loader only to give the graphic system the appropriate resources. To share graphic memory, I here use the possibilities of dataspaces. Each application that wants to use a graphical output, has to ask the graphic system for a graphic buffer. The graphic system would then find a free area and create a new dataspace on top of it. It then tells the ioguard that the application is entitled to use that dataspace. After a confirmation, the application can start using the buffer for its graphical user interface.

5.2 Virtualization

In a virtualized environment, where a virtual machine monitor (VMM) wants to provide ACPI support for its virtual machines (VM), the VMM needs to provide the ACPI hardware interface to the VM. This interface consists of the main-memory datastructures, several event registers and an interrupt. In the first part I will show a rather simple use of this interface, whereas in the second part, I show a more sophisticated example.

5.2.1 Virtual DSDT

As one main goal of ACPI is to provide a device enumeration space, that is the first service a VMM presents to its VM via ACPI. It therefore needs a tree of valid ACPI tables, containing a DSDT for the device tree. The registers and the interrupt are not needed for this example. Nevertheless the ACPI tables contain addresses of required registers. However their functions are not needed. The VMM must provide registers, but they must not do anything. The register addresses and the upper tables in the table tree are static, but the DSDT has to reflect the current device configuration.

The arrangement of devices in a VM has to fit the demands of the owner of this machine. These machines shall vary in computing performance as well as in network or disk performance. These machines shall fit the customer wishes and possibilities. Therefore the device configuration of these VMs is unknown until the contract is made. After the demands for devices are determined a VM configuration can be assembled.

A fully virtual machine has every device virtualized. These emulated devices are slow and therefore not always acceptable. Techniques have been invented to grant a VM access to a physical device. So in a device configuration of a VM there will be virtual and physical devices.

The DSDT is used to tell the VM about the device configuration. Thus the VMM has to generate a DSDT from the device setup. This generator component will be the first task for an ACPI virtualization work. The input will be a system device configuration, consisting of virtual devices and physical devices.

For each device the system device configuration might possibly contain the complete specification. To create a DSDT that does not contain any driver implementation a simple device object consisting of an identifier and a resource list will be enough. If a physical device shall be used, the VMM will get the information from the ioguard. If the generator is another component, the VMM might tell the generator directly the ioguard device id.

After the DSDT is generated the VMM has to take care how the rights for the resources are given to the VMs. They talk a hardware protocol and will not ask the ioguard to map them any resources. Additional the VMM has to tell the ioguard, which devices each VM is allowed to access. Therefore it has to interact with the driver loader and tell it to create appropriate capabilities. Afterward a component of the VMM has to request the resources from the ioguard and grant them to the VMs.

5.2.2 Map device policies to hotplugging

With the approach in the previous section I could create a VM with the current policy granting that VM a set of physical devices. If the policy would change, meaning the VMM has to remove or add devices from the VM the software on that VM might malfunction. Server hardware and software exist that can manage the plugging or removing of devices through their hotplugging capabilities. The idea arose, if one could virtualize hotplugging and apply it according to policy changes. This section will investigate the role of ACPI in such an environment.

Most interesting for this setup are PCI devices, CPUs and RAMs. Each device type has different interface to the operating system. ACPI provides interfaces to these devices, too and can play a vital role in hotplugging.

The first idea, to change the DSDT according to the new device tree does not work. ACPI has no means to detect changing tables. Also software implementations are unlikely to read the tables again after having them parsed once. Nevertheless ACPI provides support for hotplugging. In that it is restricted to devices that may be present or not, but whose existence is known in advance. Each device that may appear in the VM's configuration must be already present in the DSDT.

Due to this limitation the VMM has to generate a DSDT with all devices that the VM shall ever contain. In this device tree all nonexistent devices have to be marked as not present. The ACPI software will read the status of all device objects and thereby evaluate the present bit. It will then only load drivers for devices that are present. If the policy changes, the status of the affected device has to be changed and the ACPI software has to be informed of the device-insertion event.

To support this mechanism the ACPI hardware interface needs to be more sophisticated. The device object's status ought to be variable. It is returned by a control method. In the preceding example all control methods would just return immediate values. In contrary the control method here has to read the value from a memory address. So for all hot pluggable devices the VMM must provide a register stating the current presence status. To notify the operating system of a device insertion event, the VMM has to send the ACPI SCI, the system control interrupt. After receiving this interrupt the VM needs to determine the origin of this interrupt. The ACPI hardware specification defines several registers for that purpose. So the VMM has to provide such registers as well. When the VM has find out the affected device it will then request the status of this device object. Thereby the VM reads the value from the specified register. Should the device now be marked as present, the VM will examine further properties of the device. Most importantly the device's resources will be evaluated for the first time. The resources are in a byte stream format, and can be assembled from memory places, too. That way it is possible to generate addresses of resources only when needed. After that the VM's operating system will load a driver for this device and use it.

The PCI bus is responsible for PCI devices. In case a device is inserted the PCI bus signals an event to the operating system through the ACPI interrupt. Upon receiving this interrupt ACPI determines its origin from the PCI bus. The event will be the Bus Check Notification. It requests the operating system to perform the Plug and Play

reenumeration operation on the PCI bus. So the inserted PCI device will be discovered by the PCI system.

Memory devices can be reported to the operating system via the System Address Map Interfaces or as memory devices in the device tree of the DSDT To be able to remove memory devices, they have to be described in the DSDT. Then it is possible to use the device insertion mechanism described previously.

To hotplug processors the general approach might be possible as well. However there is no direct mentioning in the ACPI specification. For NUMA systems with nonuniform memory access architectures online hot plugging of nodes is possible, too. Similar to the PCI hotplug the Bus Check notification is send via ACPI. That also indicates that the operating system shall perform the Plug and Play reenumeration operation on the device tree. As processor power management is done by ACPI, too, it could be possible to use this mechanisms as hotplugging replacement. Instead of indicating a device as not present it could be possible to present it as sleeping. When trying to wake sleeping devices the VM will write to specific hardware registers. This writes will be intercepted by the VMM and can be ignored, as no physical hardware is involved. To keep the VM in a clear state the VMM might want to return appropriate failure messages. If the policy allows the VM to use one more physical CPU that CPU has to be bound to one virtual CPU. Afterward the virtual CPU can be awoken and the VM can start using it. Using the power management of CPUs it is also possible to throttle the speed of processors. Emulating this to a VM might be an alternative measure to change the performance of VM.

6 Outlook and Conclusion

6.1 Outlook

This work is a step toward a securely managed system. Because the ioguard only executes a policy, a driver loader is strongly needed. Afterward admission to the driver loader interface should be granted only to the driver loader.

To support systems that require probing, or to adapt to new systems automatically, the ioguard has to deal with probing. An implementation of my ideas in the design Chapter 3.3.1 would broaden the range of supported systems. The support of legacy drivers that attempt to probe for their device should not be a task of the ioguard. Instead, the DDE has to handle probing attempts. It should terminate accesses on wrong locations. Probes to the actual device can be passed.

Another topic closer to approach is hotplugging. If the hardware issues an insertion or ejection event the ioguard is the first to take action. The ioguard has to evaluate the event, determine affected devices and update its device manager accordingly. Afterward it has to inform the driver loader. The driver loader may then unload the unneeded driver or choose a driver for the new device. Means to receive and evaluate hotplugging events, as well as the interface to the driver loader are required to support hotplugging.

One main goal of ACPI is power management. That will be an interesting topic, too. The ioguard should probably evaluate all power management related information and present them to a power manager in a uniform manner. Or, the power manager is implemented as driver for all PCI and ACPI devices. Maybe then it is needed to restrict its accesses to these devices to the power management interface.

To use ACPICA as static configuration should be seen as an intermediate solution. The increased binary size, memory usage and processing time is not appropriate to read a file into a device database. I would prefer a simple file parser as static configuration backend. That would also be a preferred step prior to the use of the ioguard in systems with a fixed I/O architecture like embedded systems. For such systems it is reasonable to decrease the size of the ioguard's binary file as well. That requires a stronger modularity of the ACPI and PCI library. Ideally it should be possible to add or remove them in the Makefile.

The following sections discuss two extensions in more detail.

6.1.1 ACPI drivers

My current design and implementation uses ACPI to manage devices and to interact with ACPI hardware. A third important feature of ACPI is to provide an interface to devices, see Chapter 2.1.4. These possibilities are device specific and must therefore be implemented in drivers. They in turn need an interface to the ACPICA library, which

already supports these interfaces. The development of this addition to the ioguard interface should be done in combination with the porting of ACPI drivers.

A first extension could allow drivers to read device properties not reported so far. These properties have different names. Therefore a designer should consider two possibilities. Either he extends the ioguard with interfaces for all these ACPI defined functions, or he use a more general approach. In the latter case he also has to cope with the problem of different return types. With both solutions a further question is how the driver gets to know, which interfaces are valid for its device.

To allow drivers to alter settings of their device through the ACPI interface, an even more laborious task is needed. In addition to tell the driver about the names and return type of an interface one also has to tell the driver number, names and types of parameters. A more severe problem is that the names and types do not have to be stored in the ACPI tables. For all functions the types are defined only in the specification. Furthermore the ioguard should know the influence of function calls to the global system state.

ACPI can notify drivers of hotplugging or power management events. ACPICA forwards the ACPI generated interrupt to the appropriate driver by calling a previous registered callback function. This mechanism has to be implemented as part of the ioguard interface.

6.1.2 Dataspaces

Dataspaces describe data containers of possibly arbitrary size. However in dm_generic dataspaces are restricted to page aligned positions and lengths. This poses problems with the port I/O space, where pages can be byte granular. A more general dataspace interface should abstract from the page size. New members *alignment* or *type* could be added to the dataspace datastructure. That would however induce a considerable overhead in the dataspace protocol. As I/O pages are not likely to be of 4096 byte size, it might be possible to distinguish I/O from memory pages based on the size. That, however might introduce problems with erroneous page sizes resulting in an attempt to map I/O pages.

As L4Env has no I/O dataspaces, the region manager L4rm does not handle I/O dataspaces either. To support them, L4rm would have to manage I/O dataspaces, too. As they are in a different address space, L4rm must handle them in a different tree. The handling of I/O dataspaces depends on the decision about a more general dataspace interface. If such an interface will be designed, then L4rm will use it with minor adaptions. Otherwise, to handle I/O-pagefaults, L4rm would have to know the ioguard library that allows to map I/O-dataspaces.

The driver loader can restrict driver access to complete I/O regions. With dataspaces it is possible to define subsets of system reported I/O regions. In case a policy likes to restrict driver access to certain areas of an I/O region, dataspaces could be a means to do so. In my design the driver loader does not use dataspaces and the driver has no way to determine the available dataspaces for a resource. However this extension does not seem to be difficult and could be implemented when needed. Problems occur due to the redefinition of rights. The driver loader wants to restrict the right of the driver

to create dataspaces. Restricted regions may not be in a new dataspace. So the ioguard could handle driver loader dataspaces as special units that only describe regions. Or the ioguard restricts new dataspaces to regions already occupied by other dataspaces. Thus driver must be careful when closing dataspaces to not close that region forever. Also the semantics of ownership is not clear. Should the driver loader be the owner of its dataspaces, or should it donate them to the driver? To me, dataspaces do not fit for the purpose of restricting subregions of I/O resources.

6.2 Conclusion

The ioguard architecture allows policies to restrict drivers to their device. Using device discovery from ACPI and PCI, a dynamic view on the device configuration is possible. Policy managers can then create capabilities that allow access to devices. With such a capability, a driver can access exactly one device. To use resources of that device, the driver creates dataspaces, an abstraction allowing resource management through an application library. The view on the device configuration allows further applications and lays the foundation for various extensions.

Glossar

ACPI Advanced Configuration and Power Interface

PCI Peripheral Component Interconnect

PNP Plug and Play

DSDT Differentiated System Description Table

ISA Industrial Standard Architecture

IRQ Interrupt Request

IDL Interface Description Language

VMM Virtual Machine Monitor

VM Virtual Machine

Bibliography

[ACP03] *ACPI Component Architecture Programmer Reference, Revision 1.16*, 2003. `http://www.intel.com/technology/iapc/acpi/downloads.htm`. 11

[APJ+01] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill Framework for Virtual Memory Diversity. In *Proc. 6th Australasian Computer Systems Architecture Conference*, pages 3–11, Goldcoast, Queensland, Australia, January 2001. IEEE Computer Society Press. Available at `http://ertos.nicta.com.au/publications/papers/Aron_PJLED_01.ps.gz`. 16

[App06] Apple Computer Inc. *I/O Kit Fundamentals*, November 2006. 8

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition.* O'Reilly Media, 2005. `http://lwn.net/Kernel/LDD3/`. 6

[DRO] `http://os.inf.tu-dresden.de/drops`. 1

[FH06] Norman Feske and Christian Helmuth. Design of the Bastei OS Architecture. Technical report, Technische Universität Dresden, December 2006. Available at `http://os.inf.tu-dresden.de/papers_ps/bastei_design.pdf`. 1

[Fri04] Thomas Friebel. Übertragung des Device-Driver-Environment-Ansatzes auf Subsysteme des BSD-Betriebssystemkerns. Master's thesis, Technische Universität Dresden, March 2004. Available at `http://os.inf.tu-dresden.de/papers_ps/friebel-diplom.pdf`. 7, 22

[Hel01] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Master's thesis, Technische Universität Dresden, July 2001. Available at `http://os.inf.tu-dresden.de/papers_ps/helmuth-diplom.pdf`. 7, 22

[L4E] `http://www.inf.tu-dresden.de/index.php?node_id=1431`. 1

[LH99] Jork Löser and Michael Hohmuth. Omega0 – a portable interface to interrupt hardware for L4 systems. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999. Available at `http://os.inf.tu-dresden.de/~hohmuth/prj/omega0.ps.gz`. 7

[PNP] Plug and play vendor ids and device ids. `http://go.microsoft.com/fwlink/?linkid=49039`. 6

[PNP94] *Plug And Play ISA Specification*, 1994. 3

[SRH+06] Michael F. Spear, Tom Roeder, Orion Hodson, Galen C. Hunt, and Steven Levi. Solving the starting problem: Device drivers as self-describing artifacts. In *EuroSys*, pages 45–58, 2006. 7