I/O-FlexPages

on the x86-Architecture

Study Thesis
(Studienarbeit)

Jan Stöß
System Architecture Group
Universität Karlsruhe
stoess@ira.uka.de

May 31, 2002

Abstract

One of the essential issues of an operating system is address space management. For main memory, the L4 μ-kernel supports hierarchical user-level memory management based on FlexPages. FlexPages are memory objects of variable size that form an abstraction of pages of the virtual address space. To establish a management structure for IO-ports, FlexPages are not only defined as regions of the virtual memory, but also as regions of the IO address space, which is a separate address-space on the x86-Architecture. Via IO-FlexPages, management of IO-ports can be organised in the same hierarchical manner as memory at user-level. This Study thesis (Studienarbeit) details the design and the functionality of IO space management on the x86-architecture via IO-FlexPages, and presents an implementation for the L4 μ-kernel Hazelnut.

Contents

1 Introduction

From the user-level point of view, a μ-kernel has to serve several purposes: Enable trustworthy and fast inter process communication (IPC) between components, provide a flexible concept to manage the available memory and support control and protection of the underlying hardware.

The peripheral hardware is - at least in size - the biggest part of a computer system. IO-ports are a method to interface the processor with its hardware environment and can be seen as the processor's door to communicate to the peripheral devices: From the operating system's point of view, an external device is a range of IO ports, which bytes can be read from or written to.

Obviously, abuse of IO ports can easily lead to system crashes or even irreparable hardware damages. Thus, *protection* and *access control* of IO space are essential parts of an operating system.

On the x86-architecture, I/O space is a separate address space in addition to main memory. The hardware itself provides a management scheme for control and protection of IO ports: by privilege levels, access to IO ports can be granted or denied as a whole, and a bitmap can be used to control access to individual ports.

In monolithic systems like Linux, the kernel has access to the whole IO space. This all-or-nothing scheme can easily be implemented; however, it is rather unsafe, as it results in a mouse driver being able to reset the box or to manipulate the harddisk.

On the L4 μ-kernel, drivers are user-level processes. As a driver should only have access to the IO-ports it needs, the μ-kernel has to provide a concept to protect and to distribute the available IO-space at user level. Furthermore, this concept should be completely policy-free, allowing user-space components to decide themselves, who has access to IO space.

On could consider many concepts for IO protection; however, L4 has a mature concept for main memory. The basic idea is to treat IO ports *like memory*, that is, to use the structures and operations provided from main memory for IO space. The essential data structure for memory is the *FlexPage*: A FlexPage is a memory object in an address space with a given base address and a given size. FlexPages can be used to establish mappings between address spaces. Thereby, address spaces can be constructed *recursively* by *mapping* and *revoking* parts of one address space to another. Similar to normal FlexPages, **IO-FlexPages** are defined to refer to IO ports. Via

IO-FlexPages, IO space can be treated like main memory and the mapping concept can be used for IO ports as well; that is, regions of the IO address space can be *mapped* and *revoked* in the same manner as virtual memory at user level. An example is given in figure 1.

This concept ensures on one hand, that IO space can be distributed only to selected components, but on the other hand leaves policy completely outside the kernel.

This study thesis (Studienarbeit) details the design, functionality and implementation of the support for IO access and IO protection management via IO-FlexPages for the *L4Ka / Hazelnut* implementation on the x86-architecture.

To understand the mapping concept of the IO address space of the L4 μ-kernel, a short introduction into the mapping scheme of virtual memory and a description of the IO-space hardware features of the x86-architecture is given in the first section. The second section presents the extension of the model from main memory to I/O space and details the data structures and operations related to I/O space management. The last section details the implementation of the model, and how the concept was mapped to the given hardware protection features for I/O space. The environment used to test the implementation is presented in the appendix.

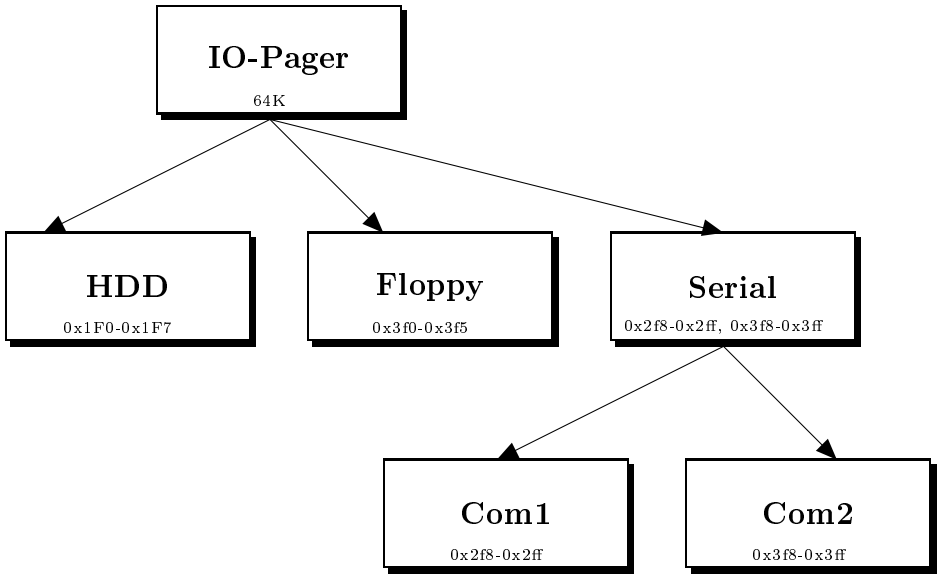


Figure 1: Hierarchical IO space

2 Background

This section provides the background to understand the mapping concept and the hardware support for IO space. The first subsection will introduce the mapping model, its data structures and its operations, and the second subsection details the IO space management on the hardware layer of the x86-architecture.

2.1 Objects And Operations On Memory

L4 supports multiple address spaces on user-level: Based on the initial address space - called σ_0 - representing the whole physical memory, address spaces can be constructed *recursively*. The pages of an address space are regions of the virtual memory and can be described by *FlexPages*. Different hardware architectures have different page sizes, and FlexPages form an abstraction to pages of virtual memory: A FlexPage describes a memory object in an address space. It has a given base address and a size; it consists of all valid virtual pages in this region. FlexPages are used to specify the source and the destination of a mapping between adress spaces: Mapping a send FlexPage to a receive FlexPage means, that the virtual memory region in the sender's address space described by the *send FlexPage* is made accessible in the receiver's address space in the *receive FlexPage*. Figure 2 illustrates that.
The mapping can be revoked by *unmapping* the given source FlexPage.

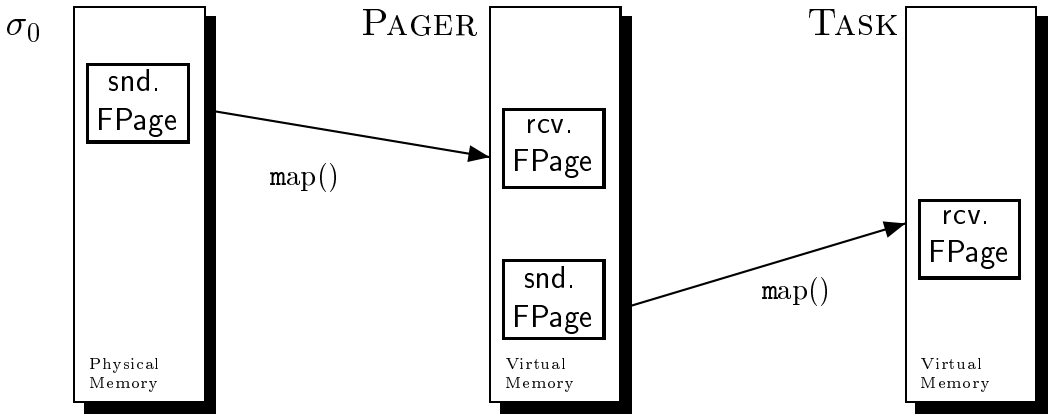


Figure 2: A Memory Mapping Example

The idea behind FlexPages is to ensure, that any mapping from the sender's into the receiver's address space only contains pages which are already mapped to the sender. This concept is flexible enough to support smooth memory management, paging or copy-on-write, but leaves the real management outside the μ-kernel.

2.1.1 Virtual Memory

To support multiple address spaces on main memory, L4 uses the translation of virtual into physical addresses. An attempt to access a location in virtual memory with no physical mapping behind will lead to a *page fault*. All physical memory originally belongs to an initial address space called σ_0, and can be mapped and revoked to other address spaces by operations described later on.

2.1.2 FlexPage Descriptors

Regions of the virtual memory can be described by FlexPages: A FlexPage has a given base address and a size, which is a power of 2.
This definition is taken from [Lie99]:
A FlexPage of size 2^s has a 2^s-aligned base address, i.e. $b \bmod 2^s = 0$. On the x86 processors, the smallest possible value for s is 12, since hardware pages are at least 4K. [...] A FlexPage with base address b and size 2^s is denoted by a 32-bit word:

fpage $(b, 2^s)$

2.1.3 Operations On FlexPages

To support mapping and revoking from one address space into another, basically three operations on FlexPages are supported by the μ-kernel:

map: maps the memory region in the sender's address space described by the specified send FlexPage into the receiver's address space. The specified receive FlexPage indicates the memory region, where the source FlexPage is mapped to. The memory region afterwards is accesible in the receiver's address space on the address given by the receive FlexPage. The sender's address space however remains unchanged. This operation is *synchronous* and *bound to ipc*.

grant: The semantics of grant is similar to map, except that the sender revokes his own rights to access the memory regions.
It is *synchronous* and *bound to IPC*, too. It is required only in special situations (See [Lie95]).

unmap: revokes all given mappings referring to the memory region described by the FlexPage.
unmap is *asynchronous* and an own system call; every task can revoke formerly given mappings from its own address space without agreement from clients. The clients receiving mappings agree *implicitly* to this fact.

2.1.4 σ_0

σ_0 is the initial address space owning all available memory at system initialization. It maps the available memory tasks requesting it, and only once. The idea behind σ_0 is, that the memory can be distributed at boot time to all initial user-level memory managers. If a page already has been mapped, subsequent requests of this page to σ_0 will automatically be denied.

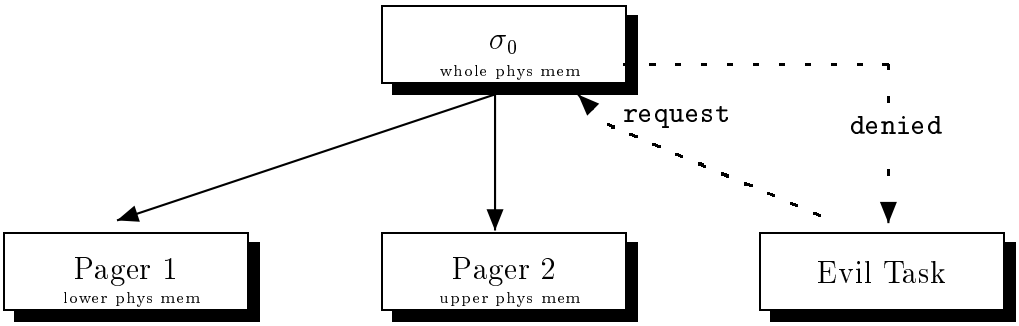


Figure 3: Memory Distribution Based On σ_0

2.2 IO-Space On the x86-Architecture

This subsection details the properties of IO space and IO instructions on the x86-archtitecture, and explains the hardware support to protect IO ports.

2.2.1 IO instructions

On the x86-architecture, IO space forms a separate and distinct address space in addition to main memory address space. Peripheral hardware is connected to the processor in the IO address space, and ports can be used as well to transmit data as to control the particular devices.

The size of the IO-space is 64 KByte, consisting of 64K 8-bit IO ports, which are individually addressable. A port is denoted by a 16bit word; the processor can transfer data from and to IO ports by four instructions:

`in(port)`: read from a port to register
`out(port)`: write to a port from register
`ins(port)`: read from a port into memory
`outs(port)`: write to a port from memory

The IO instructions move data from or to IO ports. The number of bytes to read or write can be specified from 1 byte up to 4 bytes. The source destination can be the general purpose register `EAX` or a memory location respectively. The `rep` instruction can be used to repeat the string IO instructions `ins` and `outs` several times. There exists no virtual to physical hardware translation for IO-ports.
Further information on IO instructions is available in [IA32-1].

2.2.2 IO protection

In protected mode, IO protection can be established by two mechanisms, the IO Privilege Level (IOPL), and the IO Permission Bitmap (IOPBM). In contrast to virtual memory, access violations on IO-ports will not result in a pagefault, but in a *general protection exception* (GP).

2.2.2.1 Protection Via Privilege Levels

Every piece of running code has a *current privilege level* (CPL). The CPL is a number between 0 and 3; user-level code runs with a CPL of 3, kernel level code runs with a CPL of 0. To support IO protection, every piece of

code has an *IO privilege level* (IOPL), a number between 0 and 3, too. The IOPL defines the CPL *allowed at most* to access IO-ports without any further check. The CPL normally is equal to the privilege level of the code segment, from which instructions are fetched at the moment. The IOPL is located in the `EFLAGS` register. The IOPL can be used to permit access to IO ports as a whole. On any IO access, the processor checks, if $CPL \leq IOPL$. If so, the IO operation is permitted without any further checking. If not, the processor will check the IO Permission Bitmap.

2.2.2.2 Protection Via IO Permission Bitmap

The IO permission bitmap (IOPBM) is a feature to give access to individual IO ports. Every task has its own IOPBM. The bitmap is located in the *task state segment* (TSS) for the currently running program. Every IO byte port is represented by one bit in the IOPBM. The size of the IOPBM is variable, up to 8 KByte (representing 64K ports). The IOPBM can be used for protection of individual ports. If a task performing an IO operation on a certain IO-port is less privileged (i.e. if the privilege level check fails), the processor checks all bits corresponding to the IO-port in the task's IOPBM. For a doubleword (4 Byte) access, for instance, the processor will check the 4 bits corresponding to the four port addresses.

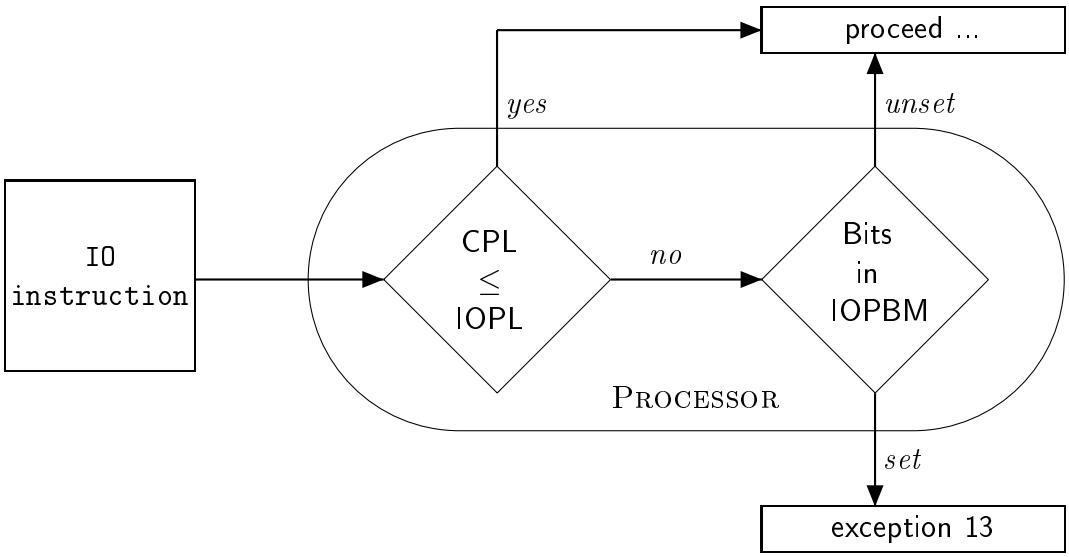


Figure 4: Processor Checking IO Access

3 The Mapping Model For IO-Ports

This section details the issues in using the mapping concept for IO space. The basic goal is to treat the IO address space like memory and to use the mapping model to enable protection and access control for IO devices. For this purpose, FlexPages have to be defined for IO space. They will be called IO-FlexPages hence. Instead of inventing new system calls for IO space, the operations `map`, `grant` and `flush` are used to handle both main memory *and* IO ports. Necessarily, these operations need to know, which address space is meant in each particular case. However, only a 32-bit word, denoting a FlexPage, is passed to the operation to identify the memory portion to map, to grant or to unmap, and every possible 32-bit word identifies a valid memory FlexPage.

Obviously, a possibility would be to add address space identifiers to the function definitions. This would require a redefinition of the operations, leading to a new specification of the L4 API. This is considered to be too complicated, firstly as the L4 API specification is cross platform, and secondly because it would require a reimplementation of already existing software running on top of L4.

The solution we pursued is to use a particular range of 32-bit words to denote operations on IO ports. Therefore, a certain range of kernel space is used to indicate operations on IO ports. Of course, these addresses cannot be used anymore to denote main memory portions.

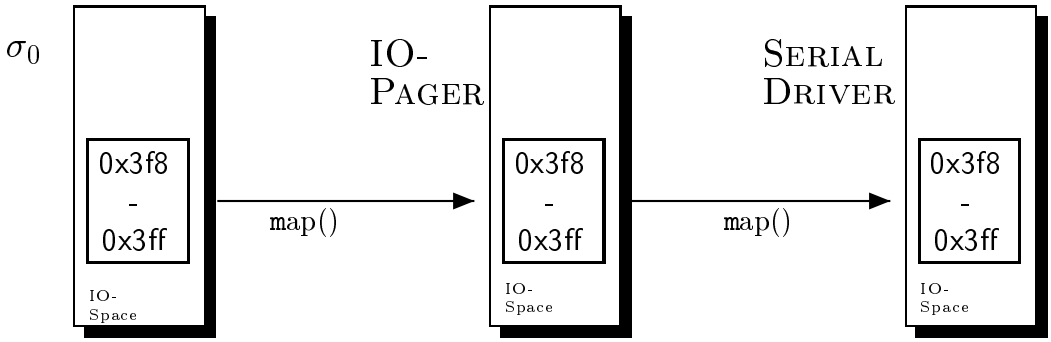


Figure 5: An IO Mapping Example

The following subsections will firstly define the IO-FlexPage and describe the functionality of the operations `map`, `grant` and `flush` concerning IO

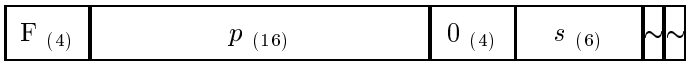
ports in detail. Further on, the extended behaviour of σ_0 is presented, and an RPC protocol is invented to handle IO requests between tasks and σ_0. The current IO-FlexPages implementation provides two different initial task states concerning their ownership of IO ports. This will be presented finally.

3.1 IO-FlexPage

The basic goal is to treat IO-ports like memory, i.e. to use the same operations **map**, **grant** and **unmap** for IO-ports. For this purpose, FlexPages are defined for IO-space:

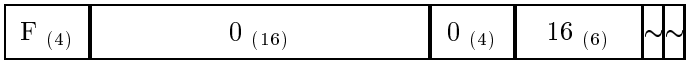
An IO-FlexPage is an object with a variable size referring to a range of IO-ports. An IO-FlexPage has a size, which is a power of 2, and a 16-bit wide base address p, referring to a range of IO-ports. An IO-FlexPage with port address p and size 2^s is denoted by a 32-bit word:

IO-fpage $(p, 2^s)$



The smallest possible value for s is 0, the highest is 16.

64K IO-Space $(0, 2^{16})$



Note, that an IO-FlexPage can have a base address not aligned to the size. An IO-FlexPage consists of all valid ports inside its domain. [1]

In fact, handling IO-ports by FlexPages results in utilizing a part of the virtual memory of each address space to denote operations on IO Ports. As mentioned, this model avoids adding address space identifiers and allows IO port management without modifying the specification of L4 X.0.

3.2 Operations On IO-FlexPages

The operations **map, grant** and **unmap** support handling IO-ports with basically the same semantics:

[1] An IO-FlexPage with ceiling above 64 KByte will consist of all valid ports, that is, all ports below 64 KByte. (Effectively, such an IO-FlexPage could be needed. Think of mapping the ports from 0xFFFA to 0xFFFF).

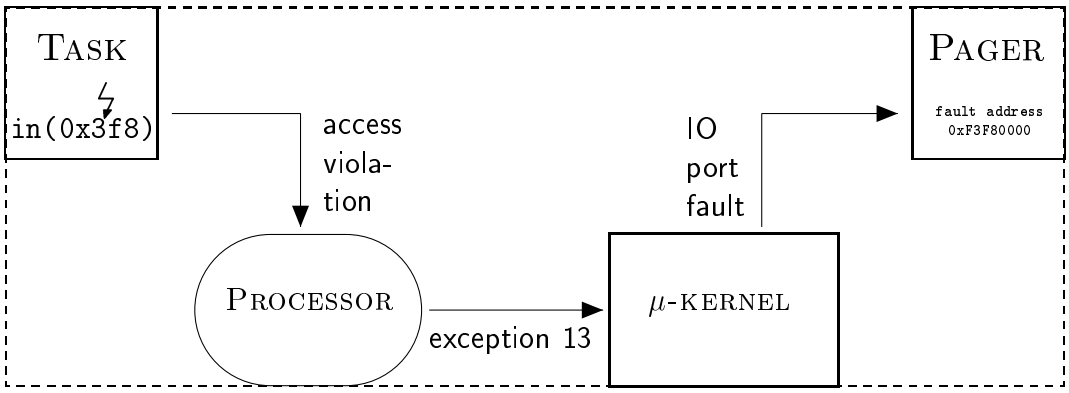


Figure 6: Translating the Exception into a Pagefault

map: maps the region of IO-ports described by the IO-FlexPage to the receiver. Previous mappings inside the IO-FlexPage will be unmapped beforehand. As there is no virtual to physical address translation for IO-ports on the x86-architecture, the mapping *always is idempotent*. Therefore, the whole IO-space (64K) should always be specified as the receive FlexPage.[2]
Since partial access rights are not supported by the hardware, too, the receiver always will have full (i.e. read and write) access to the mapped IO-ports.
The sender's address space remains unchanged.
The operation is *synchronous* and *bound to ipc*: a general protection exception due to missing IO access rights is translated by the μ-kernel into an *IO port fault* on the IO-FlexPage, which is refering to the IO-port. This page fault is brought to the pager via IPC. Thereby the pager can determine the address and the size of the IO-port the client tried to read from or write to (Figure 6). The pager then handles the IO port fault like a normal memory page fault; either by mapping or granting an IO-FlexPage or by ignoring the IO port fault.

grant: Similar to map, except the sender revokes his own rights to access the IO-ports. Like `map`, grant is *synchronous* and *bound to IPC*.

unmap: revokes all given mappings by the task referring to the IO-ports described by the IO-FlexPage. `unmap` is an *asynchronous* system call.

Note, that the μ-kernel does not map memory and IO ports simultaneously within one map message. That is, mapping the whole memory address

[2]In fact, if referring to IO Space, the receive FlexPage will be ignored completely, except that it has to be a valid IO-FlexPage.

space does *not* cover the IO address space. IO ports can only by mapped if addressed directly by their IO-FlexPage descriptors.

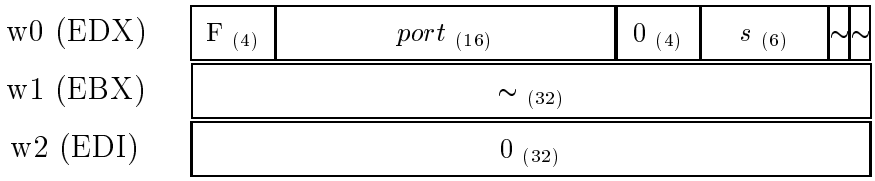
3.3 σ_0

σ_0 distributes IO-ports in the same way as virtual memory: the first arbitrary task claiming IO-ports will get them. The idea behind this is: As σ_0 denies subsequent requests on IO-ports, initial IO-pagers can request the IO-ports they want to manage; thus, distribution and control of IO-space is handled by the initial IO pagers.

3.4 RPC Protocol For IO-Port Faults

As illustrated in figure 6, if a task wants to request IO space from its pager, it can perform an IO instruction to raise an IO port fault. The message sent to the pager denotes an IO-FlexPage, whose port and size is given by the attempted IO instruction. The pager can grant access to the implicitly requested IO-ports. However, since the maximum size to read or write by `in` and `out` is 4 bytes, the maximum size of bytes to request through the μ-kernel by an IO port fault is 4 bytes. Therefore, instead of raising a port fault by an IO instruction, a task can send an *explicit IPC* to the pager, wherein the requested port and size are specified. According to the memory pagefault protocol in [Lie99], the following IO port fault RPC protocol can be used to implement explicit IO requests:

To Pager:



Intended Action: For reply, the pager should map the range of IO-ports specified by the port and the size 2^s.

The initial pager σ_0 has been modified to implement the IO port fault RPC protocol described above. That is, if a thread sends an IO request according

to the IO port fault RPC protocol, σ_0 will map the requested IO-ports. If the IO-ports are not available anymore, σ_0 replies a 0-word instead.

3.5 Legacy Support

To support unmodified "legacy software" running on top of a μ-kernel with IO-FlexPages, two different initial states of tasks can be configured at compile time of the L4 μ-kernel:

Implicit IO-Mapping: Implicit mapping of all 64k IO-ports to a task on its creation by its pager, iff the pager has full access rights itself; to deny a task to access IO-ports, the ports have to be unmapped *explicitly* in the tasks IO address space.
Implicit mapping is nested and depends on the IO permissions of the creating task: only tasks with full permissions to all IO-ports can create new tasks with full permissions to IO-ports, whereas a task lacking or having only partial permissions to IO-ports creating a new task will result in the new task having no access to IO-ports at all.

No implicit IO-Mapping: Every task has *no access to IO-ports* by default. Access to IO-ports can be given through mapping and granting. This is the more intuitive mode, as a task's initial state concerning access to IO-ports is like the state concerning virtual memory: no mappings at all.

4 Implementation

First of all, an issue implementing IO-FlexPages is to use the given hardware features to protect IO-ports. The x86-architecture supports the protection of IO-ports by two mechanisms: privilege levels and the IO permission bitmap. By privilege levels, access to IO-ports can be generally permitted or denied to user-level tasks. The IO permission bitmap (IOPBM), a per task data structure, can be used to handle access to individual IO-ports. The first subsection details, how the the hardware features were enabled and how the mapping concept is implemented at hardware layer.

IO protection by the hardware does not only affect `in` and `out`, but also clearing and setting the interrupt flag by `cli` and `sti` are protected by the IOPL. That is, only tasks with an IOPL numerically greater than the CPL

are allowed to perform `cli` and `sti`. This a rather unwanted side effect, as user-level drivers are not allowed to control the interrupt flag anymore. A possible solution to this problem is presented in the second subsection.

Obviously, the hardware does not now anything about multiple address space and their construction, and does not provide features for *hierarchical* IO protection. The mapping model requires that given mappings of IO-ports can be revoked subsequently. For this reason a mapping database is needed to keep track of all given mappings, thus enabling, that IO space can be mapped *and* unmapped properly. The design of the database is presented in the final subsection.

4.1 Implementing The Hardware Aspects

4.1.1 IO Privilege Levels

Since privilege levels only permit or deny access to IO-ports in general, the only user-level task with an IOPL of 3 is σ_0. All other user-level tasks have an IOPL of 0; thus, the processor will check the IOPBM every time a task tries to access an IO-port, excluding σ_0.

4.1.2 IO Permission Bitmap

To protect the available IO Space, every task must have its own IOPBM. It seems to be appropriate to use 8 KByte for every task, in order to support control and protection for all 64K ports.

The IOPBM itself is located in the task state segment (TSS). The x86-architecture supports a hardware-based task switch, causing a reload of the TSS and thus a reload of the IOPBM. However, L4 does not use the hardware task switch, as it is much more expensive than a well designed software switch. In L4, all tasks share the same task state segment.

The TSS is located in the kernel space of every task, which is, of course, a part of the virtual memory. Virtual memory is translated to physical memory by the memory management unit (MMU). For every page in virtual memory of a task, the MMU parses the task's *page tables*. To share the TSS among different tasks, the page table entries of the TSS point to the same physical page frames in memory. Thus, by default, all virtual addresses of the IOPBM point to a default bitmap in physical memory. Depending on the configuration of L4 (see section 3.5), the default bitmap is initialized with 0,

which means access to all IO ports, or 1, wich means no access on all ports.

The following trick makes it possible to use different permission bitmaps for different tasks without using the slower hardware task switch: To assign different IOPBMs to different tasks, *retaining* however the rest of the TSS, the bitmap is placed on an address aligned to the size of a hardware page (4KByte). The pagetable entries for the IOPBM are set to different page frames for different tasks, whereas the pagetable entry for the rest of the TSS points, as before, to the same destination. By this means, the task state segment remains at the same *virtual* address, but the *physical* frames for the bitmap are different. Thus, different tasks can have different bitmaps while still sharing the rest of the task state segment.

Note, that the IO permission bits are not cached by the hardware. Thus, if pagetable entries are exchanged due to a task switch, the processor always will parse the correct bitmap.

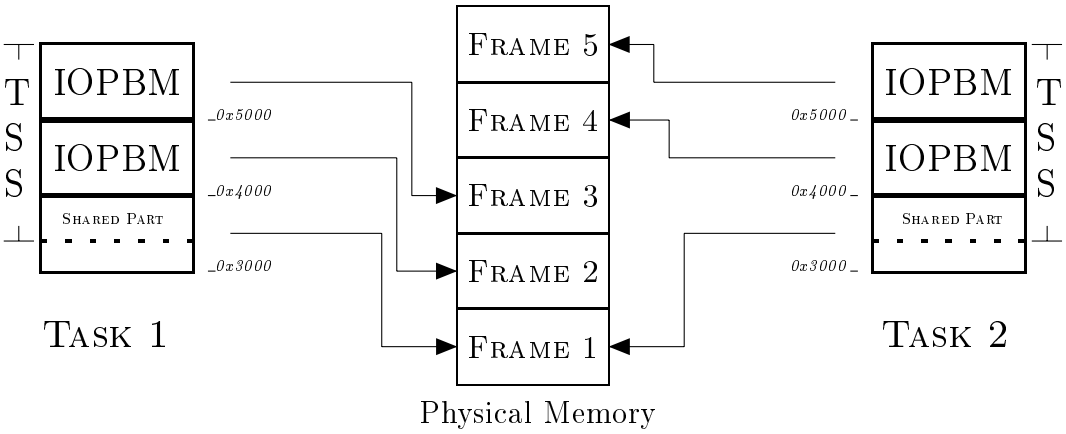


Figure 7: Same TSS, Different IOPBMs

Implementing the mapping concept at hardware layer tackles the following issues:

- From the hardware perspective, to map, to grant or to unmap IO space is nothing more than to change the corresponding bits in the IOPBM of the receiver's task (and in the sender's, if IO-ports are granted respectively removed from the sender's IO-space).

- On an exception 13, the μ-kernel has to determine the reason; if the reason was lacking IO permission, the μ-kernel has to translate the

exception into an IO-port fault, and to send that fault to the pager of the current thread.

4.2 The cli/sti Problem

`in` and `out` are not the only instructions protected by the privilege levels CPL and IOPL, but also `cli` (clear interrupt flag) and `sti` (set interrupt flag). If the IOPL is less than the CPL, an attempt to execute `cli` or `sti`, will result in a *general protection exception (GP)*.
This implies that any task, whose IO-ports are individually protected by its CPL/IOPL and its IOPBM, cannot clear or set the interrupt flag anymore. Since clearing and enabling interrupts at user level could be needed, for instance for synchronisation methods or for drivers in general, support for both IO protection and disabling/enabling interrupts is an issue of the μ-kernel. A solution to the problem is to enable *protected mode virtual interrupts*.

4.2.1 Protected Mode Virtual Interrupts

By enabling protected mode virtual interrupts (PVI), the operating system can virtualize masking and unmasking interrupts in an inexpensive way. Protected mode virtual interrupts can be enabled by setting a flag (PVI flag) in a processor specific register.

4.2.1.1 Normal hardware interrupt handling: Normal hardware interrupt handling is quite simple; clearing the interrupt flag (IF) by `cli` causes the processor to ignore all external hardware interrupts. After the interrupt flag is reset by `sti`, the processor begins to respond on external maskable interrupts again. However, as mentioned, `cli` and `sti` cause an exception, if executed in user mode (CPL=3) and the IOPL is less than 3 and if protected mode virtual interrupts are not enabled. Of course, the exception could be used to emulate the interrupts completely in software. As exceptions are expensive, however, we preferred to use the PVI approach.

4.2.1.2 Protected mode virtual interrupt handling: When the PVI flag is set, the current code is running in user mode (CPL=3), and the IOPL is less than 3, `cli` and `sti` will not cause an exception anymore. Instead, they clear and set the virtual interrupt flag (VIF), leaving the real interrupt flag (IF) unaffected.

The behaviour of the normal interrupt handling can be simulated by modification of the interrupt and exception handlers:
A task running in user mode executes `cli` to indicate, that it does not want to be interrupted now. This clears the VIF flag. If an interrupt occurs, the processor calls the interrupt handler, since the real interrupt flag (IF) was not cleared. The invoked handler checks the state of the virtual interrupt flag (VIF). If it is clear, indicating, that the active task does not want to have interrupts handled at the time, the interrupt handler sets another flag, the virtual interrupt pending (VIP) flag, and returns to the task. If the task finally resets the virtual interrupt flag (VIF) by `sti`, the processor checks the VIP flag. If the flag is set, it will automatically call the exception handler. Then the exception handler can handle the pending interrupt(s).

4.2.2 Interrupt And Exception Handlers

To explain the concept behind protected mode virtual interrupts, the modified interrupt and exception handlers are presented in pseudocode.

```
INTERRUPT_HANDLER()
 1  if VIF == unset then
 2        enqueue_interrupt()
 3        mask_interrupt()
 4        VIP = set
 5        return()
 6  else
 7        handle_interrupt()
```

```
EXCEPTION_13_HANDLER()
 1  switch instruction
 2        ...
 3        case sti
 4              dequeue_interrupts()
 5              unmask_interrupts()
 6              handle_interrupts()
 7              VIP = unset
 8        ...
```

In PVI mode, the processor automatically calls `handle_interrupt` on an

ocurring interrupt. If the user level task does not want to be interrupted, it saves the occuring interrupt, indicates the pending interrupt by setting the VIP flag, and returns to the task. As the VIP flag is set, enabling the interrupts will raise an exception 13, and the handler finds the reason to be `sti`. The pending interrupts are handled at this time.

Further information on protected mode virtual interrupts can be obtained in [IA32-3].

If a task switch occurs from a task with cleared VIF flag to a task with set VIF flag, for instance if the pagefault handler is called due to a pagefault, the pending interrupts are handled by the μ-kernel before switching to the new task.

4.2.2.1 pushf / popf

`pushf` and `popf` can be used to save the EFLAGS register on the stack respectively to pop the EFLAGS from the stack. If the EFLAGS image on the stack has a set IF flag, the IF will be set implicitly by `popf`. This is a method used frequently (for instance by Linux) to set the interrupt flag to the old value. Unfortunately, `popf` cannot be virtualized completely: if running in PVI mode, `popf` neither sets or clears the VIF flag independent of the value in the EFLAGS image on the stack, nor does it cause an exception like `sti`. Therefore, `popf` has to be substituted by assembler instructions testing the VIF flag on the EFLAGS image first, and calling `sti` explicitly, if it is set.

4.3 IO Mapping Database

To enable hierarchical IO address spaces involving mapping and revoking ports, the kernel has to keep track of existing IO-mappings in order to ensure that they can be unmapped correctly afterwards. In other words, a database is needed to store the mappings. The mapping database will *store* every given mapping. If a user-level task calls the μ-kernel to unmap a given set of IO-ports, the mapping database will *restore* the associated mappings and revoke them subsequently.

Since mapping and revoking IO Ports are hierarchical operations, a tree seems to be the appropriate data structure for the mapping. Every node in this tree would represent a mapped IO-region, and every directed edge between a parent and a child node would indicate a mapping. The number of children is completely variable ($1 : n$ relationship)

Furthermore, there must be a method to reference mapnodes by a given IO-FlexPage, that is, a method returning a task's mapping nodes corresponding to a given range of IO-ports in the mapping tree.

The first subsection will discuss an approach to reference given mapnodes. The conclusions made in this subsection lead to a data structure presented in the following subsection. Finally, the mapping algorithms are presented and detailed.

4.3.1 Referencing IO Mapping Nodes

While the implementation of the mapping tree itself is quite straight forward and similar to a mapping tree for virtual we considered many possible different approaches to implement a method returning all mappings to a task inside a given region of IO-ports. For main memory, it seems to be appropriate to use mapping nodes representing a fixed size of virtual memory, as page sizes are not completely variable. (x86-architecture supports two different page sizes: 4 KByte and 4 MByte). Consequently, referencing mapping nodes associated to a certain hardware page can be done through an index of the fixed sized mapnodes, for instance in a shadow page or a hash table. Different approaches for main memory are detailed [SU98] and [Elp98]. However, the situation for IO-ports is different, the reasons for that are as follows:

- *Variable Size/ 1-Byte-Granularity*: The granularity of IO FlexPages is 1 Byte. Therefore, mappings can range from 1 Byte up to 64 KByte. Using a fixed node size in the mapping tree would lead to a huge mapping tree. For instance, 1 byte mapnodes would waste 65536 mapnodes for a 64K mapping.

- *Predictable mapping behaviour*: In most cases, distributing IO-space is done initially, at boot time of an OS. Furthermore, a sensible handling of IO-ports would grant dedicated access to IO-ports to dedicated tasks (drivers) only. Also, the mapping tree is expected not to be very deep.

For this reasons, completely flexible sized mapping nodes were favorized for the IO mapping database. This implies, that a shadow page table or a hash table cannot be used to reference mapping nodes: a given region of IO-ports can be represented by a mapping node with a completely different address and size - for instance, searching for the mapping node representing one byte

in a tasks IO-space could return a mapping node representing the whole 64K IO-space. The most simple solution is to hold the mapping nodes of a task in a sorted linked list. This can increase the search time, but a predictable mapping behaviour should lead to few entries in the per task linked list. Additionally, using completely flexible-sized mapnodes reduces the number of mapnodes for a task - for instance a 64K mapping takes 1 node in the mapping tree. Of course, the flexible size makes parsing of the mapping tree more complicated, and increases the code size. This should be traded off by the lower run time memory consumption.

4.3.2 Data Structure For The Mapping Database

The mapping tree consists of **io mapping nodes**. As well they are used to examine the IO-ports mapped to a task. An **io mapping node** contains the following members:

`io_mnode_t{`

`taskID`	ID of the task, which the node is associated with
`lo`	Lower bound of the associated range of ports
`hi`	Higher bound of the associated range of ports
`depth`	Tree depth in the mapping tree
`prev, next`	Previous and next mapping node in the mapping tree
`prev_st, next_st`	Previous and next mapping node in the list of mappings to the task
`parent`	parent mapping node

`}`

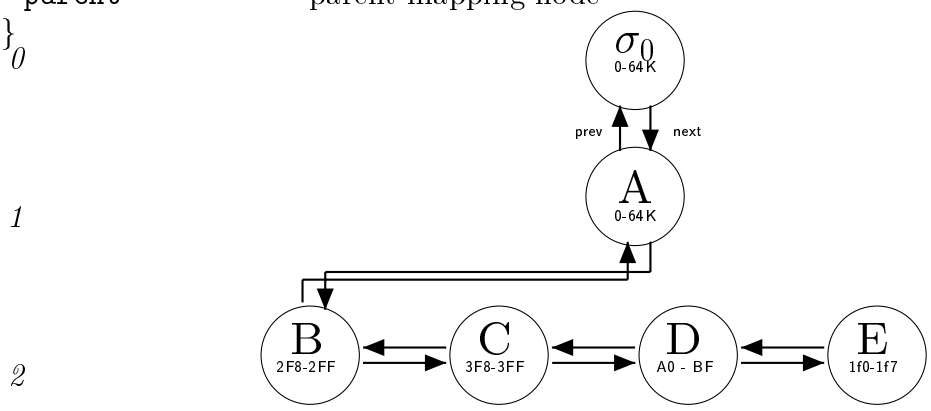


Figure 8: Database Subtree

The mapping nodes are both held in the mapping tree and in the respective thread list.

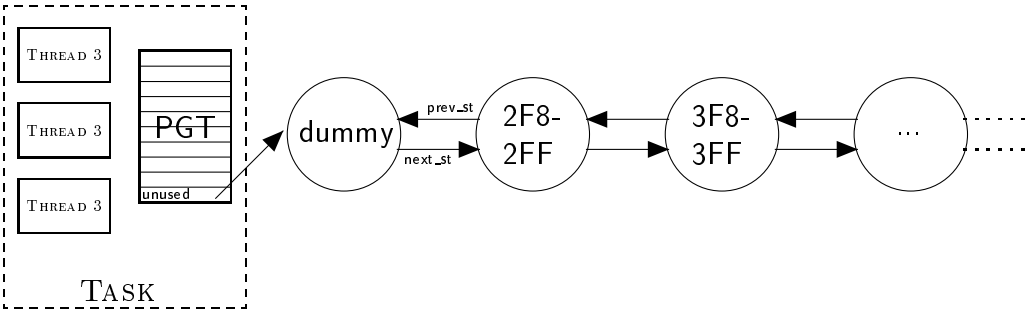


Figure 9: Task List

Instead of using a pointer per child, the tree depth and the `next` and `prev` pointers are used to determine the actual location of a node in the tree.[3]

The task list is a doubly linked list sorted by the given lower and higher bound of the nodes. Its entry is a dummy node; as threads of the same task share their IO-space, an unused page table entry of the task is used to point to dummy node marking the beginning of the task's IO-space (See figure 9). The dummy node avoids the check, if the pagetable entry has to be changed during insertion of a new node. To make parsing of a thread easier, the last mapping node in a task list points again to the dummy node. The pointer to the parent is used to prevent cyclic mappings: For instance, the mapping database has to check, if the receiver of a mapping tries to remap the IO-space back to the sender. Such a mapping has to be ignored, otherwise the map algorithm would run into an endless loop. For this purpose, the thread IDs of all recursive senders' parent nodes are compared with the receiver's thread ID. If any of them matches, the mapping will be ignored.

4.3.3 Mapping Algorithms

The following section describes the implementation of `map`, `grant`, `flush` for IO-FlexPages. Varying in size, nodes are not only allocated and removed, but in some special cases the they have to be resized or split correctly. This is somewhat complicated, in fact, it should not be needed too often. The algorithms are presented by simplified pseudocode and its explanation.

[3]Actually, the `next` pointer would be sufficient to parse the mapping tree. The `prev` pointer is needed to get the previous node, if a task unmaps parts of its own IO address space.

map: Mapping an IO-FlexPage means to map all valid IO-ports mapped inside the FlexPage in the sender's IO-space to the receiver. Previous mappings in the receiver's IO-space have to be unmapped beforehand.[4] Mapping an IO-FlexPage consists of four major steps:

1. Search the sender's mapping nodes starting from the first valid mapnode (line 1) for those relevant for mapping the given IO-FlexPage. A mapnode is relevant, iff the intersection between the represented IO-ports and those of the IO-FlexPage is not empty (lines 2,3).
2. New mapnodes have to be created under the relevant mapnodes (line 4). The actual size represented by the new mapnodes is the respective intersection.
3. Former mappings inside the intersection have to be unmapped in the receiver's IO space, including its own IO-space (line 5).
4. The new mapnodes have to be inserted in the subtree under their respective parents and in the receiver's list of mapnodes (lines 6, 7).
5. Finally, the receiver's IOPBM has to be updated (line 8).

```
MAP_IO_FPAGE(from, to, fpage)
 1  for m in from.io_mapnodes
 2        i ←[m.lo, m.hi] ∩ [fpage.port, fpage.port + fpage.size -1]
 3        if i ≠ ∅ then
 4              c ← create_io_mapnode(i)
 5              unmap(to, i)
 6              insert_in_tree(c)
 7              insert_in_task_list(to, c)
 8              zero_iopbm(to, i)
```

[4] This behaviour differs from main memory, but is consistent with the specification [Lie99]. The reason can be found in the implementation: If previous given mappings would be ignored, the map algorithm had to place new children in the subtree around the old mapnodess.

grant: granting an IO-FlexPage is very similar to mapping; several major steps can be evolved:

1. Search the sender's mapnodes starting from the first valid mapnode (line 1) for those relevant for granting the given IO- FlexPage (lines 2,3).
2. Unmap previously given mappings out of the receiver's IO-Space (line 4)
3. Grant the subtree below the mapnode to the receiver (line 5), i.e. take the mapnodes in the subtree and transfer the parts relevant for the granting operation (i.e. the instersection) to the receiver. Granting a subtree is illustrated by figure 10.
4. unmap the IO-FlexPage in the sender (line 6).
5. Update the sender's and receiver's IOPBM (lines 7, 8)

```
GRANT_IO_FPAGE(from, to, fpage)
1  for m in from.io_mapnodes
2        i ←[m.lo, m.hi] ∩ [fpage.port, fpage.port + fpage.size -1]
3        if i ≠ ø then
4              unmap(to, i)
5              grant_subtree(m, i, from, to)
6              unmap(from, i)
7              set_iopbm(from, i)
8              zero_iopbm(to, i)
```

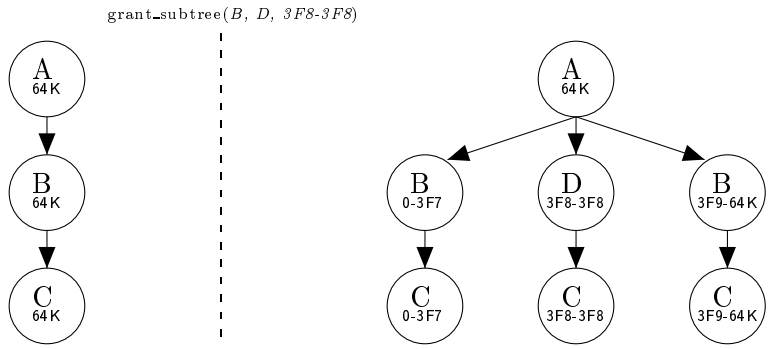
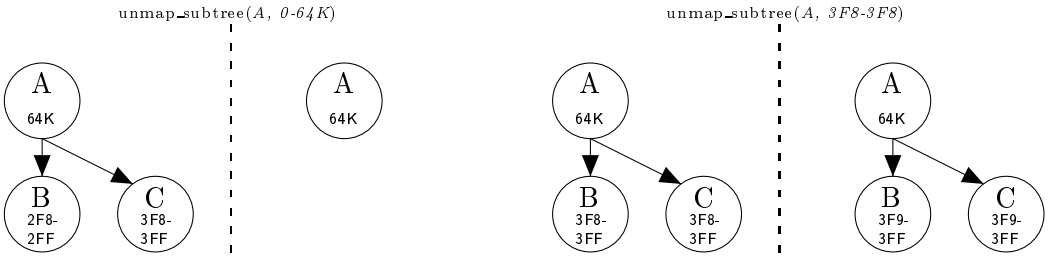


Figure 10: Granting A Subtree

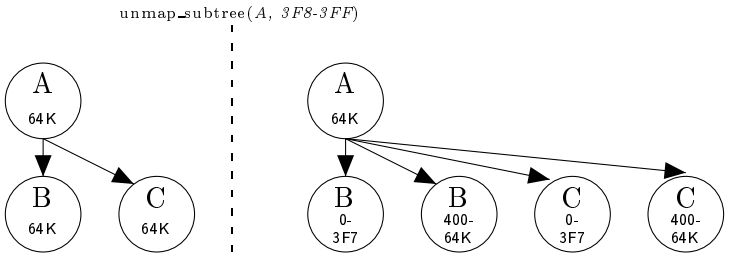
unmap: This function gets a parameter named *mapmask*, indicating, if the IO-FlexPage is to be unmapped out of the own IO-space, too. The function calls the helper function *unmap_subtree*, which is also used by map and grant. The following steps and the pseudocode detail that:

1. Search the sender's mapnodes starting from the first valid mapnode (line 1) for those relevant for mapping the given IO FlexPage (line 2,3).
2. unmap the whole subtree under the respective mapnode (line 4). As the nodes are flexible sized, the nodes possibly are not removed completely (figure 11.a) out of the subtree, but only resized (figure 11.b) or split (figure 11.c).
3. The parameter *mapmask* is passed to *unmap_subtree*, and the called function has to remove, resize or split the subtree root, too, if mapmask is true (line 4).



11.a: removing nodes

11.b: resizing nodes



11.c: splitting nodes

Figure 11: Unmapping A Subtree

```
UNMAP_IO_FPAGE(task, fpage, mapmask)
1  for m in from.io_mapnodes
2      i ←[m.lo, m.hi] ∩ [fpage.port, fpage.port + fpage.size -1]
3      if i ≠ ∅ then
4          unmap_subtree(m, i, mapmask)
```

4.3.4 Mapping Database Memory Management

The only data structure used in the IO mapping database is the `io mapping node`. The IO mapping database simply uses the memory allocation routines provided by the main memory mapping database.

5 Results and Conclusion

Since mapping, granting and unmapping of IO-space is far from being time critical, the implementation focussed on functionality and feasibility, not on speed.

With IO-FlexPages, the code size increases by about 6 KByte. Every task with specific access rights to IO ports will need a bitmap. For default access rights, the default bitmap is used. Thus the runtime memory consumption for the bitmap is 8 KByte per driver task, which is inevitable. The mapping database needs 28 Bytes per mapping. The dummy node is created for every task, thus, 28 Bytes per task are needed additionally.

A running L4Linux effectively needs to access about a hundred different ports. If every port is mapped in a single operation, this leads to a memory consumtion of about 3 KByte. Mapped as the whole 64 KByte IO space at once it will need only one mapnode of 28 Bytes. However, since access to ports is restricted to drivers and few other hardware related processes, there should not exist too many mappings in a running system. In this case, the memory consumption besides the IOPBM is more or less neglectible.

Handling IO-space is a crucial part of an OS, because access to IO ports can be abused easily. Therefore, it should be granted only dedicated and trusted components. By IO-FlexPages, IO-ports can be distributed and protected like main memory. Using the same concept for IO-space, systems designed to run upon the L4 μ-kernel can be extended easily to handle IO-ports without too much additional code. Pagers can be used to handle both main memory and IO-ports. Moreover, IO port faults can be used to determine, if faulting or evil components try to access ports they should not have access to. Thus, IO-FlexPages can lead to better protection and more transparency in systems running upon L4.

The software is available in the CVS tree. To enable IO-FlexPages support, switch on the configuration variable CONFIG_IO_FLEXPAGES in the configuration menu. Bug reports are welcome of course.

6 Future Work

Further work has to be done for the protected mode virtual interrupt handling. The current version provides only a rudimentary handler, aiming to show, that the concept works. Aspects such as potential priority problems

on task switches in PVI mode have not been examined. Furthermore, the current implementation of IO FlexPages disregards the experimental SMP support in Hazelnut.

A Testing

The IO-FlexPages implementation has been tested with L4Linux 2.4. The initial pager σ_0, the resource manager (RMGR), the Linux pager, and the Linux system calls `ioperm()` and `iopl()` have been modified to work with IO-FlexPages:

Boot Time

With enabled IO-FlexPages support, the boot sequence of L4Linux running on Hazelnut will perform the following steps to handle the IO space:

- Having started the μ-kernel and σ_0, the resource manager (RMGR) requests the whole 64K IO-space from σ_0.
- Thereafter, RMGR boots L4Linux. During initialisation, L4Linux will attempt to access IO-ports, which directly leads to an IO port fault.
- The pager of the kernel will handle the first IO port fault by requesting the whole IO-space from RMGR; whereupon RMGR maps the whole IO-space to L4Linux. The initialisation continues.

System Calls

Like in Linux, a task can get access IO-space by `ioperm()` and `iopl()`, two system calls restricted to the superuser. Running original Linux, `ioperm()` modifies the IOPBM of a task, while `iopl()` sets the IO privilege level. Running L4Linux, the result of `ioperm()` and `iopl()` is the same, the modified implementation and functionality is explained shortly:

int **ioperm**(*unsigned long* from, *unsigned long* num, *int* turn_on)

will modify the task specific, L4Linux-internal IO permission bitmap, like in Linux. In L4Linux, the size of internal bitmap is 8KByte, not 1 KByte. If turn_on is 0, indicating the access to be to the specific ports to be allowed, L4Linux will zero the respective bits in this bitmap. If turn_on is 0, indicating the access to be denied from now on, Linux will set the bits in the bitmap. Afterwards, a FlexPage containing the ports to turn off will be unmapped by L4Linux.

int **iopl**(*unsigned long* level)

As L4Linux is not allowed to change the real IOPL anymore, the behaviour is simulated: `iopl()` sets a task specific variable to the given IOPL, and the pager can map the whole IO-address-space if the variable is 3.

IO Port Faults

If the process raises an IO port faults on a given address, L4Linux kernel will check the L4Linux-internal IOPL and IOPBM. If the IOPL is 3 or the adjacent bits in the bitmap are zeroed, the kernel will map an IO-FlexPage to the task.

References

[Lie99] J. Liedtke: L4 Nucleus Version X Reference Manual (1999)

[Lie95] J. Liedtke: On μ-kernel Construction (1995)

[SU98] S.Schönberg, V.Uhlig: μ-kernel Memory Management (1998)

[Elp98] K. Elpinstone: Virtual Memory In A 64-Bit Microkernel (1999)

[IA32-1] IA-32 *Intel Architecture Software Developer's Manual*, Volume 1: Basic Architecture

[IA32-2] IA-32 *Intel Architecture Software Developer's Manual*, Volume 2: Instruction Set Reference

[IA32-3] IA-32 *Intel Architecture Software Developer's Manual*, Volume 3: System Programming Guide