Omega0: A portable interface to interrupt hardware for L4 systems

Revision: Jewel Edition, January 5, 2000

Jork Löser Michael Hohmuth

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
email: jork@os.inf.tu-dresden.de, hohmuth@inf.tu-dresden.de

Contents

Preface to the Jewel Edition of this report

We have prepared this revision of our report in response to feedback we have received during the "Workshop on a Common Microkernel System Platform" in December 1999. We added a short discussion about an alternative interface proposal that Lars Reuther and Kevin Elphinstone outlined during the workshop (Section 3.5), we have emphasized Omega0 as an *interface* (which can be implemented as a server or a library), and we corrected two errors present in the workshop version of this report. The factual errors did not invalidate the main point of our report. Please refer to the revision history in Appendix A for information on what exactly has been changed.

We uphold that the L4 community should standardize on the Omega0 interface as a pragmatic, simple-to-use, simple-to-implement, robust, portable, and efficient IRQ-delivery solution, and we suggest implementing it as a server (although a library implementation is also conceivable).

Jork Löser, Michael Hohmuth Dresden, January 5, 2000

1 A user-mode IRQ-logic server

Writing device drivers for L4-based systems so far has been a discomforting experience. L4 kernels on different platforms export slightly inconsistent hardware-interrupt abstractions, noncentralized handling of interrupt logic results in hard-to-track bugs, and hardware dependencies lead to nonportable code.

In this report, we define a portable interface to interrupt hardware, called Omega0 interface, and we propose to implement this interface in one user-level server centralizing all interrupt-logic handling. The Omega0 interface hides all hardware dependencies. It also hides the L4 kernel's interrupt interface, isolating drivers from future changes to this interface.

Omega0 implementations can enforce any interrupt-handling policies they want. For example, an Omega0 implementation might wish to hide hardware priorities of interrupt requests, ensuring that interrupt handlers with the highest L4-thread priority always run first, ignoring the hardware priority of their devices' interrupts. By placing the implementation of such policies into a user-mode server, the kernel can be kept policy-free and more flexible.

This report is organised as follows: Section 2 discusses the interrupt abstractions existing L4 implementations on different hardware platforms provide and names problems in using them without an extra layer between device drivers and the kernel. In Section 3 we introduce Omega0 as a user-level solution; we derive requirements for Omega0 and define its interface. Section 4 deals with x86-specific problems and shows how an Omega0 server could handle them; we show two possible policies an x86-based Omega0 might want to enforce, and how these policies can be implemented. Section 5 concludes this report with directions for further investigation.

Implementation	**Architecture**	**IRQ lines**	**Interrupts**	**IRQ sources**
L4/Alpha	21164	19 +	4	3
L4/MIPS	R4600	20 +	8	5
Gauntlet	SA-1100	120	32[a]	31
L4/x86	x86	16	16	15
Fiasco	x86	16	16	15

[a] The StrongARM SA-1100 CPU maps the 32 first-level IRQs to only two interrupts, but the IRQ controller that distinguishes between 32 IRQs is integrated with the CPU core.

Table 1 Number of IRQ sources on different L4 platforms. All implementations export exactly one IRQ source for each interrupt, except for kernel-internally handled IRQs like system's timer interrupt.

2 State of the art

2.1 Definition of terms

Let us start our discussion by defining a number of basic terms that are often mixed up:

Interrupt: In this report, the term interrupt refers to an asynchronous event caused by hardware that suspends normal processing in the CPU and temporarily diverts the flow of control through an "interrupt handler" routine.[1]

IRQ (interrupt request), device interrupt request: A signal a hardware device asserts on an IRQ line to inform a handler program running on the CPU about a specific hardware event.

IRQ line: The channel a device uses to signal an IRQ. The IRQ often is funneled through IRQ logic that multiplexes multiple IRQ lines before it is forwarded to the CPU. IRQ lines can be shared by multiple devices.

IRQ logic maps each IRQ line to a specific interrupt; it is possible that multiple IRQ lines are mapped to the same interrupt.

2.2 IRQ-to-IPC translation

All versions of L4 support translation of interrupts to IPC messages sent to threads running on user level. In this report, we will call these user-level threads "IRQ threads." The messages appear to come from sources with special IDs, hereafter called "IRQ sources." The L4 implementations we know of export exactly one IRQ source per interrupt, except that they do not export IRQ sources for interrupts the kernel handles itself (see Table 1).

There is a one-to-one mapping between IRQ threads and IRQ sources: IRQ threads can attach to at most one IRQ source, and IRQ sources can be assigned to most one IRQ thread. As it is possible that multiple devices share one IRQ line, and that multiple IRQ lines map to the same interrupt, IRQ threads potentially must handle multiple devices.

IRQ threads can attach to and detach from IRQ sources at run time. The kernel enforces the one-to-one association rule mentioned in the previous paragraph. It also ensures that once an IRQ source is in use, no other thread can attach to that IRQ source until its current user detaches from it. However, there is no safe way to transfer the use-relationship to another thread. The resource manager program commonly used on x86-based L4 implementations, Rmgr, works around this problem by attaching to all IRQ sources and only detaching from an IRQ source when an authorized IRQ thread requests it. This method does not completely solve the problem, however. Since release and assign are not a single atomic operation, there is a time window in which a malicious program can occupy the IRQ source.

2.3 IRQ acknowledgement

While the L4 Reference Manual [4] covers IRQ-to-IPC translation and attachment to IRQ sources, it does not cover IRQ acknowledgement at all, and as a result, all L4 implementations do it differently.

Generally, IRQ acknowledgement is divided into two parts. The first part is to inform the triggering hardware device about a successful IRQ acceptance. The second part covers acknowledging the IRQ at the IRQ logic. While communication with the triggering device must be handled by the device driver (as it is device-dependent), IRQ-logic com-

[1] Some CPU manuals define the term "software interrupt" to refer to software-triggered traps and exceptions. In this report, we disregard software interrupts, and the term "interrupt" always refers to interrupts triggered by hardware.

munication is the same for all devices, but depends on the hardware platform and its configuration.

L4/Alpha. The Alpha architecture has two levels of IRQ logic. The first level is the logic connected directly to the CPU that triggers one of 4 interrupts. The second level consists of logic connected to each of the 4 first-level IRQ lines. The boards supported by L4/Alpha have two cascaded PC-compatible legacy programmable interrupt controllers (PICs) with 16 second-level IRQ lines on first-level line 0, a PCI controller with one second-level line on first-level line 1, and a clock (with no extra IRQ logic) on line 3; line 2 is currently unused.

Like all other L4 implementations, L4/Alpha [6] exports to user level the distinctive feature seen by the CPU: one IRQ source per interrupt. Following the rule that the kernel should handle all devices it hides behind abstractions, the first-level IRQ logic is acknowledged in the kernel. The second-level hardware, that is, the PICs and the PCI-IRQ logic, must be handled by the IRQ thread attached to the interrupt in question.

L4/x86. The x86-based IBM-PC architecture uses two cascaded 8259A PICs to implement its interrupt logic (see Figure 3). Each PIC manages 8 IRQs. One IRQ is used for cascading, so 15 different IRQs are available to the system. Including the IRQ used for cascading, they are numbered from 0 to 15. The PICs can raise a different interrupt for each IRQ number.

L4/x86 [4] exports one IRQ source per interrupt. Unlike L4/Alpha, the kernel does not handle IRQ acknowledgement at all; the IRQ threads must know how the kernel has programmed the PICs, and must acknowledge IRQs accordingly.

On most machines, each device uses a different IRQ line. In the PC architecture, each IRQ line generates a different interrupt. As each interrupt corresponds to one L4 IRQ source, all IRQ sources map to a single device. That's why, device drivers hitherto have used the kernel's IRQ-message mechanism directly. This mode of operation has meant that all device drivers need to acknowledge their IRQs themselves, requiring synchronization between the driver threads to avoid interference between PIC accesses in different drivers. Also, there are subtle dependencies between the hardware priorities of different IRQs and the order in which they are acknowledged (explained in more detail in Section 2.4), leading to the requirement that the IRQ threads' software priorities must be ordered according to the hardware priorities of the IRQs they handle.

In practice, the synchronization of IRQ threads is difficult to get right, especially when IRQ threads run in separate server tasks. The required knowledge about the PIC mode the kernel has set up results in bugs and code which in not portable to a future multi-processor version of L4 (because SMP PCs use a different PIC mode) or to setups where multiple devices share one IRQ line.

Fiasco. The Fiasco µ-kernel [1] runs on the same hardware as L4/x86, and it is binary compatible with L4/x86. Therefore, by default it emulates L4/x86's behavior, and inherits L4/x86's problems as well. In fact, Fiasco reinforces the problems because it is a preemptible kernel. When an interrupt has occurred, L4/x86 does not allow further interrupts to occur until the IRQ thread handling the interrupt has been scheduled. Where this feature hid many synchronization bugs under L4/x86, these bugs popped up happily and numerously under Fiasco because Fiasco likes to enable IRQs for better real-time qualities.

In addition to L4/x86-emulation mode, Fiasco also implements an experimental version of in-kernel IRQ acknowledgement that follows L4/Alpha's "the kernel should handle all devices it hides behind abstractions" philosophy. In this mode, IRQ acknowledgement at the PICs is centralised in the kernel. Drivers do not have to care about acknowledgement, hardware priorities, synchronization, or PIC mode at all.

The in-kernel IRQ-acknowledgement mode has been implemented without altering the kernel interface. It works in a quite simplistic manner that limits its usefulness somewhat, as follows: When an interrupt occurs, an IRQ message is queued for the IRQ thread that attached to it; at the same time, the IRQ is acknowledged with the PIC, but also masked, so that this particular IRQ cannot occur again until unmasked. When the IRQ message is actually delivered to the IRQ thread, the IRQ is unmasked again so that it can occur again. This works fine for nonperiodic, edge-triggered IRQs and IRQ threads that cope with new IRQ messages as soon as they wait for new messages. It does not work for level-sensitive IRQ lines (i. e., all shared IRQ lines): The IRQ would be signalled again as soon as it is acknowledged because the driver has not yet programmed the device to deassert the IRQ. Please refer to Section 4.3 for strategies that do not have these limitations.

2.4 IRQ priorities

On all L4 platforms, IRQs have hardware priorities. When an IRQ occurs, further IRQs with a lower hardware priority than the first one are blocked until the first IRQ has been acknowledged. If hardware priorities are in a different order than the software priorities of the assigned IRQ threads, two interesting problems emerge: priority inversion and incorrectly-sequenced IRQ acknowledgement.

When an IRQ thread with a high L4-thread priority attached to an IRQ with a low hardware priority cannot operate because another IRQ thread with a lower thread priority has not yet acknowledged its IRQ with a higher hardware priority, we have a case of classical priority inversion (see Figure 1).

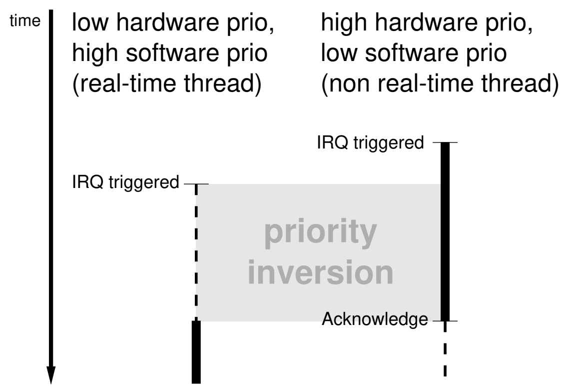

Figure 1 Priority inversion. A high-priority IRQ thread (left) is blocked because a low-priority IRQ thread (right) has not yet acknowledged its IRQ.

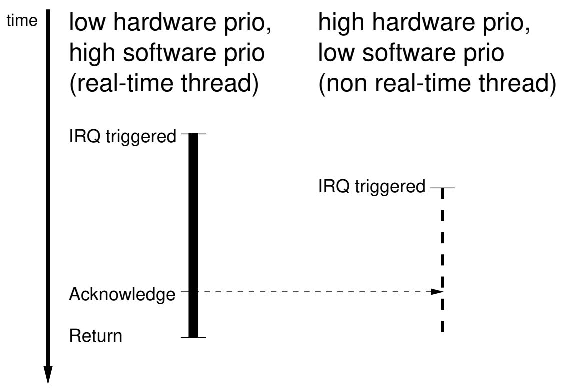

Figure 2 Incorrect acknowledgement sequence. In this example, a high-priority IRQ thread (left) wants to acknowledge its own IRQ, but accidently acknowledges the wrong IRQ because the low-priority thread (right) has not got a chance to acknowledge its IRQ.

The only way to prevent priority inversion and to decouple thread priorities from hardware priorities is to acknowledge IRQs as early as possible. However, this is difficult to achieve without a central high-priority instance that handles IRQ acknowledgement: The low-priority IRQ thread might get preempted by other, higher-priority threads before it gets a chance to finally acknowledge its IRQ.

In particular on the x86 platform, there is also the danger of acknowledging IRQs in wrong sequence. Acknowledge operations on the PICs are not IRQ-specific and always refer to the latest IRQ (which, by definition, has the highest hardware priority). Under certain conditions, IRQ threads can accidently acknowledge a wrong IRQ. This can lead to system hangs. Figure 2 illustrates this problem.

The problem can be avoided by sorting the IRQ threads' priorities according to their IRQ's hardware priorities. If IRQ threads run in multiple unrelated tasks, this requirement leads to inflexible systems with "priority inversion by design."

2.5 The problem

In the previous subsections, we have identified the following problems:

1. While all L4 implementations consistently export one IRQ source per interrupt, not all implementations drive the IRQ logic triggering the interrupt that was translated into an IPC.

2. L4 does not offer a way to safely transfer an IRQ-source attachment from one thread to another.

3. There is no standardized way to link multiple device drivers to one IRQ line.

4. Noncentralized handling of IRQ logic leads to hard-to-track synchronization bugs, nonportable code, and priority inversion.

In this report, we will not propose changes to the *kernel* interface to solve Problems 1 and 2. However, our proposed *user-level* solution to Problems 3 and 4, presented in the following section, will also work around Problems 1 and 2 as a side effect.

3 Omega0: The IRQ-logic server

3.1 A user-mode server

There have been some heated discussions in the L4 community about the IRQ-related interfaces the kernel should export, and the results so far are not satisfying. We would tend to recommend that all implementations should behave like L4/Alpha (devices hidden behind kernel abstractions are also driven by the kernel); however, this model would mean that the kernel implements policy instead of just mechanism, sacrificing much of the flexibility exercised in Section 4. More work is needed to define a kernel interface that is consistent between all L4 implementations and that is flexible enough to support all sorts of IRQ logic and policy. That's why, we have decided to factor out this problem and leave, for now, the kernel–user interface as unspecified as it is.

Instead, we propose an interface to a system-central IRQ-logic server that hides *all* IRQ logic from device drivers. This interface is supposed to be portable between architectures. The server implementing this interface will be architecture-specific and implementation-specific as long as no standard kernel interface has been defined. (This situation is similar in nature to the relationship between Sigma0 and the kernel: For example, the location of the kernel-info page is unspecified, and there is no standard way for Sigma0 to find out where it is located; therefore, it is impossible to implement a portable Sigma0 server.)

The IRQ-logic server, henceforth called *Omega0* (or Ω_0, pronounced "omega-zero"), shall belong to the system's trusted computing base. A server like this is a novelty for x86-based L4 systems; however, such a server has been required before for L4/Alpha to demultiplex interrupts to IRQ lines.

If the kernel synchronizes accesses to the IRQ logic, and its IRQ abstraction is powerful enough, we can also imagine Omega0 being implemented as a user-level library interfacing the kernel's IRQ facilities directly. This design trades binary portability across different L4 implementations on the same platform for a performance advantage. However, in the rest of this report we will assume that Omega0 is implemented as a server.

Why should the Omega0 interface be implemented on user level—or, in other words, why do we not propose to make the Omega0 interface the kernel interface we talked about in the first paragraph of this section? Because we do not want the kernel to know about the configuration of second-level IRQ hardware. In fact, an implementation of Omega0 may want to configure the IRQ logic in a specific way, as illustrated in Section 4. The kernel should not need to care about what happens as long as it can still handle its timer interrupt and forward interrupts as IPC messages to Omega0.

3.2 Problems with this design

Implementing Omega0 as a user-mode server has four important drawbacks.

First, device drivers must find out or know the task ID of Omega0. If we do not want to give Omega0 a well-known task number like Sigma0's, then drivers have to rely on a naming service to find Omega0. It is out of the scope of this report to discuss name services, but there are obvious and simple solutions on all platforms. In the future, we might want to agree on a standard name-service interface.

Second, there is an additional overhead for each delivered IRQ because it must be piped through another thread running in a separate address space, resulting in more address-space and protection-domain boundary crossings. However, the additional cost is smaller than the cost for manipulating the IRQ logic. This is true on all architectures, even on the x86 with its relatively expensive system calls and address-space switches; the cost for the latter can be remedied using the small-address-space mechanism of L4/x86 [3].

Third, IRQ messages to device drivers now become subject to message redirection using the clans-and-chiefs mechanism [4], potentially leading to costly message redirects through chief tasks. Currently, this is not a problem because on all L4 systems we know of, the device drivers run within the root clan anyway, and in the future the mechanism set to replace clans and chiefs [2] will provide an efficient solution.

Finally, Omega0 may fail to notice when a driver thread is deleted, preventing new attachments to IRQs forever. This problem is closely related to rights management: How does Omega0 determine if an IRQ-attachment request that possibly conflicts with already-existing attachments should be honored, and when should Omega0 delete an attachment without the consent of a client? We suggest to postpone this problem for now. Solutions to these problems exist and can easily be implemented (for example, a simple cookie-based capability scheme combined with a timeout and a probe to detect dead drivers could be used; we have reserved IRQ number 0 for sending a probe event to the driver), but we should not standardize on one solution before we have discussed system-wide rights-management schemes that apply to more services than just IRQ-logic handling. This discussion is outside the scope of this report.

3.3 Requirements

Omega0 should provide clients with a simple user interface similar in spirit to the Linux kernel's `request_irq` interface:

- It should export an "IRQ line" abstraction, that is, a different IRQ number for each individual IRQ line a device can be attached to, as opposed to an "interrupt" abstraction L4 exports.
- It should completely hide the IRQ logic from device drivers.
- A device-driver thread should be able to attach to more than one IRQ line at a time. This can be useful when probing for a device.
- Multiple device drivers can attach to one IRQ line, facilitating shared IRQ lines. When an IRQ occurs on that line, each registered driver is notified in turn until one of them "consumes" the IRQ.
- Drivers can "mask" the IRQ they are attached to. This operation prevents further IRQs to be delivered on that line and possibly other lines (with a lower hardware priority) too, depending on the configuration of the IRQ hardware.
- The interface should be optimized for the common case, that is, no more than one RPC should be necessary to consume an IRQ and wait for the next one.
- Interrupt numbers do not have to be contiguous, but can be organized hierarchically (e. g., IRQ numbers 1–0x10 for ISA interrupts, 0x1001–0x1020 for PCI interrupts). That's why, the interface shall support enumeration of all interrupt numbers.
- RPC messages related to IRQ delivery should fit into a short-message IPC, even on the register-impaired x86.

3.4 Omega0 interface

In this section we will define an RPC interface for communication with the Omega0 server. Even though IRQ notifications are asynchronous in nature, the interface is synchronous in that Omega0 clients have to explicitly request every single IRQ notification using the `request` RPC.

In the following subsections, the RPCs are defined in terms of C data structures and a CORBA-like interface description. This definition needs to be mapped to C bindings and an L4-IPC "wire" protocol for each L4 platform. We define the mapping to correspond to the current mapping implemented by the port of the Flick IDL compiler to L4 as defined in [7]. This way, all RPCs with the exception of `pass_attach` fit into register-only "short" IPC messages even on the x86 platform.

All RPCs except the `request` RPC should be sent to (and wait for a reply from) local thread 0 of the Omega0 task. The `request` RPC should be sent to the server thread returned by the `attach` operation.

As Omega0 needs to start an IRQ thread for each interrupt, but has to pass to its clients IRQ notifications for potentially multiple IRQ lines mapped to different interrupts using only one thread ID (the one returned by `attach`), an efficient implementation of Omega0 depends on the availability of a "Send-As" IPC function that allows IPC-sending threads to pretend to be a different thread of the same task. Liedtke has proposed such a function before and has also implemented it in an experimental version of L4 [5]. We hope that the L4 community standardizes on an interface for this function soon.

3.4.1 Client connect and disconnect

```
typedef struct
  {
    unsigned i_num      : 16;
    unsigned i_shared   :  1;
    unsigned i_reserved : 15;
  } omega0_irqdesc_struct_t;

typedef union
  {
    omega0_irqdesc_struct_t s;
    unsigned i;
  } omega0_irqdesc_t;

void attach(in omega0_irqdesc_t desc,
            out int lthread);
```

Attaches the current driver thread to an IRQ line. The `i_num` element specifies a valid IRQ number (one that `list_first` or `list_next` returned); this is not a bit vector, but an integer. The `i_shared` flag denotes the driver's willingness to share the line with other drivers handling other devices on the same IRQ line. All other bits (15 on 32-bit architectures) are reserved; in the future, drivers may want to define CPU affinity or suitability as a source for randomness.

When a driver newly attaches to an IRQ line, Omega0 masks the IRQ line. The driver first has to unmask it using a `request` operation before new IRQs can occur on that line. This affects other drivers attached to the same line as well: Omega0 only unmasks the line if no driver has masked it.

The return value `status` is negative on failure, or the local thread number of the IPC-partner thread in the Omega0 task; this is the thread the `request` RPC should be sent to. When a driver thread attaches to multiple IRQs, it will always get the same IRQ-partner thread.

```
void detach(in omega0_irqdesc_t desc,
            out int status);
```

Detach from an IRQ line. A `status` value of 0 denotes success, a negative value means failure.

```
void pass_attach(in omega0_irqdesc_t desc,
                 in l4_threadid_t new_driver,
                 out int status);
```

Move the right to attach to an IRQ line to a different driver thread. This operation also automatically detaches the current thread if it has been attached. A `status` value of 0 denotes success, a negative value means failure.

3.4.2 IRQ handling

```
typedef struct
  {
    unsigned a_param    : 16;
    unsigned a_wait     :  1;
    unsigned a_consume  :  1;
    unsigned a_mask     :  1;
    unsigned a_unmask   :  1;
    unsigned a_reserved : 12;
  } omega0_request_struct_t;

typedef union
  {
    omega0_request_struct_t s;
    unsigned i;
  } omega0_request_t;

void request(in omega0_request_t action,
             out int status);
```

Multi-function RPC for handling IRQs; multiple options may be set in one request.

If the `a_wait` flag is set, the call blocks until one of the assigned interrupts occurs; otherwise, the call returns immediately. The `a_consume` flag tells the server that the previously reported IRQ has been consumed and may occur again.

The `a_mask` option tells the server to mask the IRQ line denoted by the IRQ number in the `a_param` element. Finally, `a_unmask` is equivalent to `a_mask` but unmasks instead of masks. Omega0 keeps track of the "masked" state for each driver attached to an IRQ line and only unmasks the line if no driver has masked it.

Setting the `a_mask` and `a_unmask` options at the same time is reserved for future extensions.

A wait operation without either a consume option (i. e., `a_wait` set but `a_consume` not set) or a previous nonwaiting consume operation (i. e., without a previous request with `a_consume` set but `a_wait` not set) tells the server that the driver cannot handle the IRQ.

If multiple drivers have attached to a single IRQ line, and an IRQ occurs on that line, Omega0 notifies all drivers in turn until one driver consumes the IRQ. It is not defined whether Omega0 waits for a nonconsuming driver's next `a_wait` request before notifying the next driver, or whether Omega0 notifies all drivers in parallel. Also, the order in which drivers are notified is not defined.

If `a_wait` was set and an IRQ occurs, the returned `status` value denotes the number of the corresponding IRQ line. Otherwise, 0 is returned on success. In any case, a negative `status` value denotes an error.

All these operations have been squeezed into one complex RPC to allow most IRQ handshakes to take place with a single exchange of short-message IPCs.

3.4.3 Enumerating available IRQs

```
void list_first(out int first_irqnum);
void list_next(in int irqnum,
               out int next_irqnum);
```

Get a sequence of valid IRQ numbers. Number 0 is reserved and never is a valid IRQ number.

3.5 Alternative interface

During the First Workshop on a Common Microkernel System Platform, Lars Reuther and Kevin Elphinstone outlined a different interface for shared-IRQ delivery: Instead of our "ping–pong" scheme where a central server sends IRQ notifications to all assigned driver threads in turn, they propose to set up chains of driver threads connected to the same IRQ line. The first thread in the list receives a notification from a central server and checks whether "its" device has triggered the IRQ. If so, it consumes the IRQ by notifying the central server; otherwise, it forwards the notification to the next thread in the list, which in turn checks whether the IRQ was destined to it. This game continues until either one driver consumes the IRQ, or the end of the list has been reached.

This alternative has the advantage of being faster than the Omega0 solution because it saves two cross-address-space IPC operations per nonconsumed IRQ notification. The improved performance comes at the cost of a less straightforward initialization sequence and reduced flexibility because drivers now need to know their neighbors in the shared-IRQ chain. Also, the interactions between the IPC partners are more difficult to program because notification passing cannot easily be mapped to procedural RPC interfaces.

We believe that the extra cost of two IPCs is in the noise of the much more heavy cost of device input–output operations drivers have to carry out in any case, and is worth spending for the simplicity the Omega0 interface gains. That's why we plead for employing the Omega0 interface.

4 X86-specific Omega0-implementation issues

4.1 IRQ logic on x86-based PCs

The x86-based single-CPU PC architecture uses two cascaded 8259A programmable interrupt controllers (PICs) to implement its interrupt logic (see Figure 3). They are used to forward IRQs from hardware devices at the input–output bus to the CPU. The PICs can trigger a different interrupt for each IRQ.

Each PIC manages 8 IRQs. One IRQ is used for cascading the PICs, so 15 different IRQs are available to the system. Including the IRQ used for cascading, they are numbered from 0 to 15. Each IRQ has a fixed set of assigned hardware devices.

PICs support hardware priorities for their IRQs. That is, while an IRQ is handled by the processor, the PICs will not signal another IRQ with the same or a lower priority until the first IRQ has been acknowledged at the PIC. However, IRQs with a higher priority will still be signalled. The IRQ priorities are defined at system-initialization time.

Every IRQ on a PIC can be masked using the *interrupt mask register* (IMR). That means that hardware can request interrupts, but if the IRQ is masked, the PIC will latch one request and will not forward it to the CPU. When an unmask operation removes the mask, the PIC will propagate latched and future interrupt requests.

Fiasco and L4/x86 use the so-called *special fully nested mode* of the 8259A PICs, which changes the IRQ's priorities and requires special interrupt acknowledgement. Special fully nested mode is used to assign the highest hardware priority to the timer interrupt at IRQ 8. This is essential for the kernel's real-time capabilities because the L4 kernel uses the timer interrupt for scheduling.

There are two ways to acknowledge IRQs with a PIC: the specific and the unspecific end-of-interrupt (EOI) operation. Unspecific EOI acknowledges the the IRQ that occurred last (which, by definition, is the highest-priority IRQ that has not been acknowledged yet). Specific EOI acknowledges one particular IRQ. Fiasco and L4/x86 use unspecific EOI; for

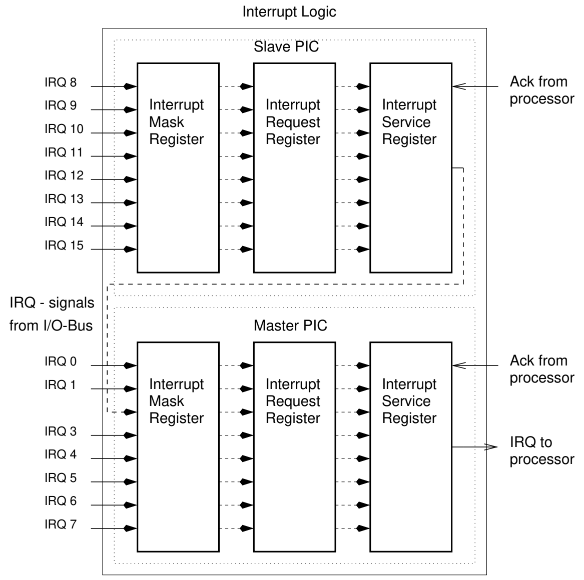

Figure 3 Programmable interrupt controllers (PICs) in the IBM-PC architecture

special fully nested mode, specific EOI seems to be unsupported or at least undocumented.[2]

IRQ acknowledgement in special fully nested mode differs from IRQ acknowledgement in standard nested PIC mode. In standard nested PIC mode, the processor first acknowledges (i. e., sends an unspecific EOI) the master PIC. If the IRQ to acknowledge was triggered by the slave PIC (i. e., IRQ $\geq$ 8), the processor acknowledges the slave PIC as well. In special fully nested mode, when acknowledging interrupts triggered by the slave PIC, the processor must additionally check if no other IRQ is pending at the slave PIC. It acknowledges the slave, and only if no other slave IRQ is pending it acknowledges the master PIC.

The 8259A PIC supports two interrupt modes: edge-triggered IRQ and level-sensitive IRQ. Level-sensitive mode is needed for shared IRQ lines, that is, if multiple devices use the same IRQ. In level-sensitive mode an IRQ on a PIC must not be acknowledged before either the driver has caused the device to deassert the IRQ line, or the IRQ has been blocked, because otherwise the IRQ would occur again instantaneously. Blocking can be achieved by masking the IRQ in question at the IMR.

The state of an interrupt line is latched in the *interrupt request register* (IRR). The IRR gets its input from the IMR which can mask interrupt lines. In level-sensitive mode an IRQ asserted by a hardware device corresponds to a certain static voltage on the interrupt line. If such a level-sensitive IRQ is masked and acknowledged while a device still asserts it, but is deasserted before it is unmasked, the PIC ensures that the interrupt will not be spuriously triggered again at the unmask operation.[3]

4.2 Requirements

As detailed in Section 2.3, the x86-based L4 implementations by default do not acknowledge IRQs with the PICs. That's why, an Omega0 implementation for the x86 platform needs to care about the IRQ acknowledgement at the PIC level.

Omega0 must ensure that the IRQs are acknowledged (1) using the correct PIC mode (special fully nested mode), (2) in a synchronized fashion, and, most importantly, (3) in the correct order (the IRQ that occurred last must be acknowledged first). Otherwise, random IRQs may be blocked forever and the system may hang.

The last requirement means that Omega0 needs to keep track of the order of IRQ occurrences. An easy way to accomplish this duty is to assign Omega0's IRQ threads priorities sorted in the order of the hardware priorities of the associated IRQs.

The synchronization requirement is easy to fulfil because *all* IRQ threads run within the same task.

4.3 Policies

There are a number of conceivable policies that Omega0 can implement.

Strategy 1: Hide hardware IRQ priorities. One possible policy is to completely decouple hardware IRQ priorities from thread priorities of device drivers that are Omega0 clients: Drivers should not be affected by other drivers with lower thread priorities, even if the other driver's IRQ has a higher hardware priority.

To disregard hardware priorities, Omega0 must acknowledge IRQs immediately after they occur. This fulfils the correct-order-of-acknowledgement requirement and allows triggering new IRQs immediately, even those with lower hardware priorities, while other IRQs are handled by user IRQ-handler threads.

However, as discussed in Section 4.1, level-sensitive IRQs cannot simply be acknowledged without first either masking the IRQ or having it deasserted by the device; otherwise, a new IRQ would be generated instantaneously. Immediately acknowledging IRQs without first masking it may result in problems with edge-triggered IRQs also if the hardware does not stop generating IRQs while the IRQ-handling driver is active.

[2] We do not have available a good PIC specification. Our documentation (PIIX4 chipset manual; legacy-PIC specification for CHRP) seems to vaguely indicate that unspecific EOI must be used in special fully nested mode, but Jochen Liedtke asserts that specific EOI also works in that mode (personal communication).

[3] Personal communication with Jochen Liedtke, Workshop on a Common Microkernel System Platform, Kiawah Island, SC, December 1999

To overcome these problems, the following method can be used: After receiving an IRQ message from the kernel, Omega0 masks the IRQ in question at the IMR. Next, it acknowledges the IRQ with the PIC. Then, it passes an IRQ notification to all devices registered to it.

Because the IRQ has implicitly been masked, it also must be implicitly unmasked next time the device driver does an `a_consume` operation unless the device driver first sends explicit `a_mask` or `a_unmask` requests.

This procedure results in overhead for masking and unmasking the IRQs on the interrupt mask register. The advantage of this solution is that the hardware and software priorities are decoupled.

Strategy 2: Low overhead. If decoupling hardware and software priorities is of less importance, double IRQ occurrence can be prevented using the following method: Immediately after an IRQ occurs, the associated driver threads are notified, but the IRQ is neither masked nor acknowledged. Omega0 acknowledges IRQs at the PICs when a driver requests an `a_consume` operation; however, this operation must preserve the correct-order-of-acknowledgement property.

Because higher-priority hardware IRQs may have been triggered while the handler was active, the IRQ must not be simply acknowledged; higher-priority IRQs must be acknowledged first because by definition they occurred later. That's why the kernel checks if higher-priority IRQs were triggered in between. If so, the current IRQ is marked for acknowledgement, but not actually acknowledged. If not, the IRQ will be acknowledged immediately. The kernel then iterates the IRQ list ordered by priorities downwards until it finds an triggered but unmarked IRQ. All IRQs that have been marked for acknowledgement earlier are acknowledged now.

Implicit masking and unmasking the IRQ is not required with this strategy. Because the IRQ is acknowledged at the PIC after the driver has "consumed" the IRQ (i. e., deasserted the IRQ at the device), no spurious double interrupts occur even for level-sensitive IRQs.

Of course, drivers can still explicitly mask and unmask IRQs. This allows drivers to mask and acknowledge IRQs early (i. e., when they have to perform more work before waiting for the next IRQ) to avoid priority-inversion effects.

5 Conclusion and future work

We have presented a new portable interrupt interface named Omega0 for device drivers running on top of L4. This interface is more than existing interfaces tailored towards the needs of device-driver writers or porters. The Omega0 interface hides hardware dependencies, making device drivers more portable and more robust.

We have argued that Omega0 should be implemented as an user-level server because that keeps the L4 μ-kernel policy-free and more flexible. There can be multiple Omega0 implementations for a given L4 platform, enforcing different policies without requiring changes to the kernel or to device drivers.

Finally, we have discussed two possible policies and how they could be implemented on the x86 platform.

In the near future, we envision further refinements and first implementations of this interface. An important precondition is an implementation of the IPC redirection and access control framework presented in [2].

A number of important topics relevant to this work must be investigated in future work: How can the interrupt-related L4-kernel interfaces be improved? How should access rights and name service be managed? How does Omega0 learn of clients that have disappeared?

Acknowledgements

We would like to thank Sebastian Schönberg and Hermann Härtig for their support and advice. We also would like to thank Alan Au, Kevin Elphinstone, Daniel Potts, and Adam Wiggins for their input on non-x86 L4 platforms.

We also would like to thank Jochen Liedtke for sharing his PIC expertise with us, and the participants of the First Workshop on a Common Microkernel System Platform, December 1999, for their feedback on this work.

References

[1] Michael Hohmuth. The Fiasco kernel: System architecture. Unpublished manuscript, 1999.

[2] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *7th Workshop on Hot Topics in Operating Systems (HotOS)*, Rio Rico, Arizona, March 1999.

[3] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[4] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sep 1996.

[5] Jochen Liedtke. L4 version X in a nutshell. Unpublished manuscript, August 1999.

[6] S. Schönberg. L4 on Alpha, design and implementation. Technical Report CS-TR-407, University of Cambridge, 1996.

[7] Volkmar Uhlig. A micro-kernel-based multiserver file system and development environment. Master's thesis, TU Dresden, October 1999. Also Research Report RC ???, IBM T. J. Watson Research Center, Yorktown Heights, NY.

A Revision history

January 5, 2000: "Jewel edition:"

- Introduction and Section 3.1: Changed to emphasize Omega0 as an API that *can,* but does not have to, be implemented as a user-mode server.
- Section 3.5: Discuss IRQ sharing using a driver chain instead of Omega0's ping–pong scheme.
- Section 4.1: Unspecific EOI might actually work in the PIC's special fully nested mode, but we don't know for sure. Also, acknowledging level-sensitive IRQs early doesn't lead to double IRQ triggering if the IRQ line was masked. Fixed Figure 3 accordingly.

December 9, 1999: "Camera-ready" version for the First Workshop on a Common Microkernel System Platform