An Architecture Independent Kernel Debugger for Hazelnut

Studienarbeit

Stephan Wagner
System Architecture Group
Universität Karlsruhe
swagner@ira.uka.de

September 14, 2001

Abstract

Supervisor: Dipl.-Inform. Uwe Dannowski

This document describes the design, the functionality, the usage and some implementation details of the new kernel debugger for the L4Ka Hazelnut micro-kernel.

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Synopsis . . . 9

2 Related work **11**
2.1 The micro-kernel approach . . . 11
2.2 The L^4 micro-kernel . . . 12
2.3 The L^4 Version X kernel Hazelnut . . . 13
2.4 Reasons for a kernel debugger . . . 14
2.5 The disadvantages of the old versions . . . 15

3 Design objectives **17**
3.1 Separate the debugger from the kernel . . . 17
3.2 Achieve architecture independency . . . 18
3.2.1 Architecture independent part . . . 19
3.2.2 Architecture dependent part . . . 19
3.3 Achieve extensibility and maintainability . . . 20
3.4 Easier usage . . . 20
3.5 Hide the debugger . . . 20

4 Implementation details **23**
4.1 Initialization of the debugger . . . 23
4.2 Interrupt handling of the kernel and the debugger . . . 23
4.2.1 IPC tracing . . . 24
4.2.2 PF tracing . . . 24
4.2.3 Exception tracing . . . 24
4.3 Thread names . . . 25
4.4 Data-types . . . 25
4.4.1 The name table . . . 25
4.4.2 The trace buffer . . . 25
4.4.3 Restrict events . . . 26

4.4.4 Trace controlling 26
4.5 Trace buffer dump 27
4.6 Implementation of event restriction 27
4.7 Configuration via Xconfig 28
4.8 Shared files between the old and the new debugger 28

5 Summary and future work 29
5.1 Summary 29
5.2 Future work 29

A Kernel debugger manual 31
A.1 Xconfig 31
A.2 The main menu 31
A.2.1 Reset 32
A.2.2 Continue 32
A.2.3 Display trace buffer 32
A.2.4 Step instruction 32
A.2.5 Step block 33
A.2.6 The sub-menus 33
A.3 The CPU menu 33
A.3.1 CPU state 33
A.3.2 Performance counters 34
A.3.3 Disassembler 34
A.3.4 Breakpoints 34
A.4 The kernel menu 34
A.4.1 Kernel-data 35
A.4.2 Priorities 35
A.4.3 IPC tracing 35
A.4.4 Exception tracing 36
A.4.5 Interrupt assignment 36
A.4.6 TCB dump 36
A.4.7 Task dump 36
A.5 The memory menu 37
A.5.1 Memory-dump 37
A.5.2 Memory-dump in other address-space 37
A.5.3 Page table dump 37
A.5.4 Page-fault tracing 38
A.5.5 Dump mapping DB 38
A.5.6 MDB tracing 38
A.6 The configuration menu 39
A.6.1 Name a thread 39

A.6.2 Nicknames for some well known threads 39
A.6.3 Enable/disable cache 39

B Key-bindings **41**
B.1 Hierarchically ordered Key-bindings 41
B.2 Shortcuts 42

Bibliography **43**

Index **45**

Chapter 1

Introduction

1.1 Motivation

When constructing a kernel or an operation system it is often very hard to determine in which state the CPU or the rest of the system currently is. And it may be much harder to find any errors in the program code because the computer simply stops on such an error or does some weird things. It is not possible to show any information about the system because the only layer except the kernel or operating system is the pure hardware which has no debug functionality.
To debug the system, that means display the state of the processor and of the currently running programs, a different program is used, the kernel-debugger. It sets up a new layer between the hardware and the software.

1.2 Synopsis

This paper consists of four major chapters. Following this introduction, Chapter two describes the basic ideas of micro-kernels, especially the L^4 kernel and the currently existing debugger. Chapter three presents the design goals and in Chapter fourare described some aspects of the implementation.
Appendix A presents a manual of the new debugger, appendix B a short summary of the debugger's keystrokes.

Chapter 2

Related work

2.1 The micro-kernel approach

The following text is (mostly) taken from [8].

The idea of micro-kernels is to keep the kernel minimal. Ideally, all OS services are implemented outside the kernel as servers that execute in user-mode in their own address-space.
An address-space on the hardware level is a mapping between virtual and physical page frames.
This approach reduces the kernel's size and protects the OS services from each other and the users. Furthermore, it also protects the user from OS services.
Then it is possible to introduce OS services that are not necessarily fully trusted by every user and by any other OS service. The central idea of this approach is the address-space paradigm. Any server (and any user-mode program) has its own private address-space.
Since file systems and the network (like TCP/IP) reside in different address-spaces, they are isolated and protected from each other. As a consequence, the micro-kernel has to supply user and system services (the kernel does not differentiate between them) with a cross-address-space communication facility. This mechanism is usually called inter-process communication (IPC).

But this approach has some other advantages too. Different application-program interfaces (API), file systems and OS personalities can coexist in the system and it is flexible and extensible. Furthermore server malfunctions are as isolated as normal application malfunctions, the system structure is modular and easy to maintain. And such systems tend to have fewer errors.

2.2 The L^4 micro-kernel

L^4 is a second generation micro-kernel developed by Jochen Liedtke at the GMD, IBM and the University of Karlsruhe.
Currently there are three implementations of L^4 for x86. Two of them implement the L^4 Version X.0 API [1], an assembler kernel and a C++ implementation with some inline assembly called Hazelnut. Another version called Fiasco by the Universität Dresden implements the L^4 Version 2 API.
The most important abstractions in the L^4 kernel are threads and address-spaces.

A thread is characterized by a register set (including the instruction pointer and the stack pointer), a status and the address space in which it executes. Furthermore it needs a thread, a unique identifier to make communication possible. These values are managed by the kernel in thread control blocks (TCB).There is a (kernel) TCB per thread.

Thread communication is a fundamental feature of the L^4 kernel. It is done via IPC. During the IPC there is an agreement between the sender and the receiver. The sender decides which information will be sent. The receiver can decide whether to interpret the received information.
With an IPC a message is delivered from one thread to another. A message consists of a string up to two megabytes, up to 31 indirect such strings or a flexpage for map messages (see address-spaces below).
The atomic IPC system-call can be parameterized as *send, call, receive* and *send and receive*. An open and closed wait is possible. IPC is an atomic operation for performance reasons.
An IPC will only take place, if the receiver has agreed. After invoking an IPC operation the thread blocks until the message has been transfered, the IPC has been aborted or the given timeout has expired.
The IPC in L^4 Version X.0 knows a send, a receive, a sender page-fault and a receiver page-fault timeout. A timeout can be 0 (do not wait), ∞ (wait forever) or periods from $1\mu s$ to 19 hours. These timeouts are stored in a 32 bit word. The timeouts are necessary to avoid denial of service attacks. If i.e. a page-fault occurs while sending or receiving a message, the corresponding pager solves it and the transfer is continued. But when the pager does not map the desired page (if it is maleficent) the IPC will never finish and the involved threads will block forever. A given page-fault timeout causes the IPC to be aborted after the specified time.

A basic idea of L^4 is the possibility to construct address-spaces recursively

outside the kernel. On system start the physical memory is represented by the initial address-space $\sigma 0$. To build other address-spaces on top of $\sigma 0$ the kernel provides three operations: map, grant and unmap.
By invoking a *map* operation the owner of an address-space can make accessible a region of its address-space to other address-spaces, provided the recipient agrees. These regions are described by flexpages (fpages). A fpage consists of all pages actually mapped in this region. The minimum size of a fpage is equal to the minimum hardware page size. The mapee itself can recursively map the page to any other address-space.
A thread can remove any of the pages it has formally mapped to other address-spaces with the *unmap* operation. The affected address-space owners agreed to a potential unmap when they received the page by mapping. So it is safe to unmap the pages without consent of the mapees.
The *grant* operation is a specialization of the map operation. By granting a fpage, the pages are mapped to the grantee's address-space provided the recipient agrees. But in contrast to the map operation the pages are removed from the granter's address-space. Since the granter has no longer access to the pages, these pages cannot be removed. This allows some memory guarantees to special threads.
The important restriction is that instead of physical page frames only pages accessible to a thread can be mapped or granted.
This address-space concept enables user level management of address-spaces. The mappings are stored by the kernel in a mapping tree (the mapping data base MDB) to be able to do the unmap operations as described above.
A description of the latest L^4 micro-kernel version is given in [2], the behavior of map, unmap and grant is explained in [8].

2.3 The L^4 Version X kernel Hazelnut

The motivation for Hazelnut was to get rid of assembler. Its goals are performance, portability and maintainability. It is written at the University of Karlsruhe under the terms of the GPL.
To achieve portability it is divided into one architecture independent and multiple architecture dependent parts. The supported platforms are ARM and x86.

Some hardware dependent control mechanisms like page-faults, the processor's clock or IO raise interrupts. On x86 all the system-calls in Hazelnut like *IPC, unmap, task_new, ...* are invoked by software interrupts (see [1], [5]). System-calls on ARM are implemented as invoking page-faults on de-

fined addresses. So the page-fault handler can decide either to jump to the coresponding system-call handler or to handle the page-fault. This is done for performance reasons, but allows the same approach.
This approach has a big advantage. It is not necessary to differentiate between system-calls and interrupts. The interrupt can be interpreted as IPC from the invoker (hardware or software) to the corresponding interrupt handler.
The following text refers to the x86 implementation of Hazelnut. After an interrupt occurred the corresponding assembler stub becomes active, prepares the stack for the handler function which is written in C++ and finally calls this function. The mechanism is described in section 3.1.
Some input and and output macros, some system-calls and some hardware controlled things like thread-switch and the initialization of the hardware are also written in assembler. The rest of the kernel is written in C++.

The kernel can be configured by xconfig, a slightly modified Linux TCL/TK configuration utility. This displays a configuration menu for the kernel and the debugger. The configuration of the debugger is described in A.1, the configuration of the kernel is not part of this paper.

For further information about the L4Ka project see [6].

2.4 Reasons for a kernel debugger

The kernel debugger enables the user to inspect the kernel state. This can be necessary to find errors in the kernel or to check its correctness after changes. Furthermore it can be useful when writing an operating system on top of the kernel. There it is often helpful to see some kernel data or events like the stack, the occurring page-faults or some details of the IPCs. This is not possible for user-mode programs and it is no required kernel functionality and therefore not wanted in the kernel (micro-kernels must be small!).
This job is done by the kernel-debugger. To do it, the debugger needs intimate knowledge of the kernel. But the kernel should be independent of the debugger (because it does not need it to work). Therefore it must not know the debugger, the debugger has to be transparent to the kernel.
And for security reasons, the debugger must be transparent to user programs as well. How this is done is shown in chapter 3.5.
The user manual for the debugger can be found in appendix A.

2.5 The disadvantages of the old versions

There are three existing debuggers for the L^4 kernel. One implemented in C++ in the L4Ka project, another one in assembler for the assembler kernel and a third one for Fiasco.
In the L4Ka version of the kernel debugger some of the debug functionality is implemented in the kernel. It remained there from the early days.This was the only way to test parts of the kernel while it was still under construction. After finishing the kernel it was fast and easy to implement the debugger by reusing this code. So this implementation of the debugger is more a prototype than a finished program.
And so it has some big disadvantages. As the kernel and the debugger are interwoven much of the code of the debugger is inside the kernel code. This enlarges and slows down the kernel which is not desirable. The kernel should be as small as possible. To solve this problem, the debugger code can be "removed" from the kernel code by compiler statements. But then the kernel and the debugger are not flexible. The entire kernel must be rebuilt to change the functionality of the debugger. In other words, it must bee known which operations the debugger should perform when compiling the kernel, or the kernel must be recompiled after a crash with the necessary functions. Then the system must be restarted and it must end up in the same state to check what has happened. This is not always possible. So the kernel can either be small and fast, or the debugger is flexible.
If the debugger is completely separated from the kernel, its functionality can be changed dynamically.
Another big disadvantage of the interweave is that the kernel knows (and uses) the debugger. The most important functions of the debugger are IPC tracing and page-fault tracing. These functions are implemented in the IPC handler and the page-fault handler of the kernel. When IPC or page-fault tracing is enabled, these handlers do the debugger's job. They collect the needed information. But a micro-kernel does not produce any output and has therefore no print method. This print method comes with the debugger. The kernel knows that there is a debugger and it knows something about its implementation. This means the debugger can not be changed without checking the influence on the kernel. And the kernel cannot run without the debugger (it ends in linker errors in the current implementation). This is not desirable on systems limited to a small amount of memory because the debugger uses memory that user programs may need.
This problem can also be solved by separating the debugger from the kernel.

The assembler implementation of the debugger is strictly separated from

the kernel. This is basically much better than the debugger in Hazelnut. But assembler programs are not portable and not easy to be expanded or maintained.

Chapter 3

Design objectives

> "The kernel debugger is much like a small OS beneath the kernel." (J. Liedtke)

The design of the new debugger tackles five goals.

- Separate the debugger from the kernel.
- Make the debugger architecture independent.
- Make the debugger easy to be expanded and maintained.
- Make the debugger easier to use.
- Make the kernel independent from the debugger.

3.1 Separate the debugger from the kernel

Section 2.5 shows, that it is not desirable to make the kernel dependent on the debugger. To separate the two parts the debug code has to be removed from the kernel and put into the debugger. But then it is more difficult for the debugger to get its information.
To reach this goal the debugger has to know the interfaces of the system-calls and the implementation of their functioning. Furthermore there must be a possibility to interrupt the kernel while handling the system-call and perform the debugger's code in between. This is called daisy chaining.
On x86 processors every system-call of L^4 is invoked by a software interrupt (as seen in section 2.3). To get the necessary information the debugger must recognize the interrupts and handle them before the kernel does.
After the interrupt has occurred the corresponding assembler stub whose

address is stored in the processors interrupt descriptor table (IDT) saves the registers, builds an exception frame on the stack. This mechanism is invoked by hardware. Then the real handler C-function is called by the assembler stub.
Such an exception frame consists of 17 dwords (32 bit values). The fault code, the values of the data, code and stack segment registers, an extra segment register, the eight general registers eax to edi, an error code, the fault address, the eflags and the user stack pointer. This information is needed by the debugger to see the status of the system when it was invoked.
To be able to trace the interrupts (and so the system-calls) a new assembler stub for the interrupt is registered in the IDT (an exact description how this can be achieved is given in the Intel manual [5]). The new assembler stub prepares the stack for the new handler and calls it. This handler displays the event or records it into the trace buffer (see section 4.4.2 and appendix A). After the handler has finished the assembler stub cleans up the stack and finally jumps to the original assembler stub whose address has previously been restored. (See section 4.2 for more details about the output of the major events.) Then the event is handled by the kernel as if the debugger did not exist.
To disable the debugger only the original IDT entry must be restored. So the debugger has no more effects on the system.
This approach has another advantage. The functions of the debugger can be turned on and off during runtime without recompiling the kernel and rebooting the system. And it also makes the kernel smaller and faster.

3.2 Achieve architecture independency

To achieve architecture independency the debugger's code has to be separated into two parts. The first part contains all the architecture specific functions and instructions, the second part contains the independent functions of the debugger. The interface between those two parts must be well defined.
To port the debugger (like the kernel) only the files depending on the architecture have to be rewritten for the new architecture. Then the kernel debugger has to be compiled. Within the current implementation of hazelnut the debugger is linked as a library to the kernel. So it is convenient to build the kernel and the debugger together. The platform can be chosen within the xconfig menu.

3.2.1 Architecture independent part

The L^4 kernel uses some abstractions of the hardware which lead to data-structures within the kernel like the TCB or the mapping database (MDB). Reading and displaying of the values of those data-structures does not depend on the used architecture.
The same applies to the initialization and handling of the debugger's data-structures like the nicknames of threads or the trace buffer.
The input functions that read from the debugger's input device (using the getc function, which depends on the hardware) like kdebug_get_hex, kdebug_get_task or kdebug_get_thread are also architecture independent.
The output function printf is also independent of the used architecture. It is able to handle all the L^4 specific data-structures and the nicknames of the threads. To display the characters on the screen, printf uses the architecture dependent function putc.
Furthermore the C-functions of the interrupt handlers are architecture independent. These handlers get the values they need as parameters from the assembler stubs and print them or enter them into the trace buffer.
Finally the menus and the navigation through them (since the debugger is written in C++) can run on any system with a respective compiler.

3.2.2 Architecture dependent part

After being called the debugger has to decide what to do. This is done by interpreting an error-code given to it as parameter. Therefore the debugger has to know the procedure call convention of the architecture.
The debugger uses daisy chaining to get its information. On x86 architectures this is done by entering the assembler stubs into the processor's IDT (see section 3.1). On ARM architectures there is also a comparable table which holds the addresses of the handlers. (For performance reasons Hazelnut's system-calls on ARM are implemented as invoking page-faults on defined addresses (see 2.3). To use daisy chaining with this method, the trace-page-fault handler of the debugger has to be modified in a suitable way.) This technique can be used on any other architecture I know.
The architecture depending part of the debugger contains some other functions, too. As shown in section 3.5 it could be desirable to disable the cache and the performance-counters.
After the debugger has finished the processor has to be in the same state as before calling the debugger. This means the debugger must clean up the registers and memory after its work.
The debugger offers some architecture depending functions to the users.

These functions are collected within the debugger's CPU menu. They enable the user to display the current state of the CPU, disassemble the next instructions, do some profiling, handle breakpoints and use the performance-counters.
The functions putc and getc which contain the layout of the debugger's input device also depend on the architecture.

3.3 Achieve extensibility and maintainability

Because of the separation of the debugger from the kernel the debugger can be easily expanded and maintained. To add or remove some functions the new code has to be written into the corresponding file without any influence on the rest of the system. The files contain all the functions and data structures corresponding to only one part of the functionality of the debugger. This makes it easy to find the place in the source code to change or to add functionality to.

3.4 Easier usage

To facilitate the use of the debugger its functions have been arranged in sub-menus. There is one sub-menu for kernel data, one for memory depending functions, one to display some information of the CPU or use some of its special functionality like the performance-counters and one to configure the debugger. So it is easier to find the desired operation.
The menus are described in appendix A, the key-bindings in appendix B.
But there are also some shortcuts to make frequently used functions accessible in all menus. A list of these shortcuts is shown in B.2.

3.5 Hide the debugger

It is basically very good to make the kernel independent of the debugger. It is important because the kernel does not need the debugger to do its work. To achieve this it is necessary that the debugger is completely invisible to the kernel and to the user-mode programs, except if they explicitely invoke it. But it is very easy to implement when using daisy chaining on interrupts. The only thing to do is enter the debugger (on x86 architectures) by invoking a software interrupt. Then the same mechanisms as shown in section 3.1 can be used for all entry points to the debugger (the main entry point or while displaying the system-calls). This section shows how to hide the debugger to

the kernel by using this technique on x86 architectures.

Running a program on a computer has some effects on the system. The program needs memory for its code and data, these are written in the caches, it needs time to execute the instructions and the performance counters might be increased. But the debugger should be transparent to the rest of the system (kernel and user programs). So the effects must be hidden if possible or at least minimized.
The debugger is entered by generating either interrupt 3 (Breakpoints or main entry point to the debugger), 2 (NMI) or 1 (debug exception). How those interrupts are handled is described in section 4.2.
The handler for these interrupts is the normal entry point to the debugger, the kdebug_entry function. It needs an exception frame which is built by the assembler stub of the corresponding interrupt.
This mechanism is invoked by hardware and cannot be influenced.
Then enter_kdebug calls the arch_entry function. Here it is now possible to "hide" the debugger. Therefore the first operation of the arch_entry function is reading the processor clock. By writing this value back to the clock register as one of the last instructions when leaving (see arch_exit below) it seems to all the other running programs, that no time has elapsed. So time virtually stops for the programs while the debugger works, the debugger actually needs no time for the rest of the system.
Then the performance counters are disabled, this means they stop counting while the debugger works. So the events the debugger raises will not be noticed by the kernel and the programs in user-mode.
In order not to change the cache content the debugger disables the cache after disabling the performance counters. The bits CD (cache disable) and NW (no write-back) are set in cr0 to do that. The effect is that the processor will not replace cache-lines after cache misses. Write hits will update the cache, but the debugger does only desired changes on kernel data or on its own data. So the effects of the debugger to the cache are minimal. For the kernel and the user-mode programs the cache content does not change. This slows down the debugger, but performance is not important for the debugger and the other programs won't get more cache misses than running without it. What exactly happens is described in [5], chapter 9, table 9-4.
The only effect of the debugger on memory is the space it occupies. But the debugger is statically linked as a library to the kernel within the kernel-space. So it is not possible that the debugger invokes any page-fault on its own code. (Otherwise the result would be the same as if a page-fault within the kernel had occurred.) The only possibility for the debugger to raise page-faults is while dumping memory of threads. To avoid this the debugger checks if the

page to be dumped is in memory or not. If not, hashes are printed instead of the memory content.
As the user-mode programs are not persistent they will not see changes of the memory consumption of the kernel if the debugger is linked to it or not after a reboot.
If this is done the processor's state is "frozen" and the debugger runs with only minimal effect on the rest of the system. Then the event that caused the debug entry is dealed with.
When the debugger has finished the original state of the processor is restored by enabling the cache and the performance-counters and setting the clock back to the restored value. This is done by calling the arch_exit function after the debugger has finished its work. After that the interrupt handler cleans up the stack and executes an interrupt return.
An exact description of the hardware programming is given in [5].

Chapter 4

Implementation details

This kernel-debugger has been implemented for x86 processors, mostly for i686. Some functions need a local APIC, so a MP-ready Pentium II or Pentium III is needed to use them (non-MP processors don't have one).

4.1 Initialization of the debugger

On boot-up of the system the debugger initializes the serial port, if the input or output device in the xconfig menu is set to *com*. Otherwise the hardware (keyboard or screen) is used.Then the entry point of the debugger is written in the *kdebug_exception* field of the *kernel_info_page* to find it later in the system.
After that the data-structures of the new debugger are initialized. This memory is allocated statically. This is necessary because the debugger (and also its memory consumption) is invisible to the kernel, so the kernel cannot decide if the memory is used by the debugger or not when the debugger allocates its memory dynamically. Then the index of the *name_tab* is set to zero and last the *trace_buffer* is created as a doubly linked list.

4.2 Interrupt handling of the kernel and the debugger

Section 2.3 shows, how system-calls and exceptions are handled by the kernel and chapter 3 shows, how the debugger handles them.
One of the main features of the debugger is its ability to show (and trace) those events. Every other interrupt can be handled by this technique as well by entering a respective assembler stub into the IDT.

The following subsections describe the major events.

4.2.1 IPC tracing

To be able to monitor the IPCs is most important for the debugger.
IPCs are implemented as invoking interrupt 30H or sysenter.
When IPC tracing is on, any IPC is monitored before it is handled. The debugger traces every IPC that meet the specified restrictions. It is possible to restrict IPCs to up to five specific threads and to threads whose thread-id is inside or outside an interval. Up to five specific thread-ids could be excluded but by now this resets the other restrictions. It is also possible to monitor only IPCs with a send part like *call, send and reply* and *wait*.
The sender and the receiver, the send descriptor and receive descriptor, the three message words in the registers, the timeouts and the instruction pointer of the sender, the local time and the values of the two performance counters are displayed.

4.2.2 PF tracing

Page-fault monitoring is also most important for the debugger.
If a page-fault occurs the hardware generates an interrupt 14H.
So like IPC tracing, the page-faults that meet the restrictions are monitored before they are handled, if page-fault tracing is on. Page-faults can be restricted to a list of up to five specific threads and to threads with thread-id inside or outside an interval, or to all threads not in the list. The last setting will reset the other restrictions.
For the monitored page-fault it is traced either if it occurs in kernel or in user mode, the thread-id, the fault address, the instruction pointer and the pager of the thread, the error-code, the address of the current page table, the cpu on which the page-fault occurred the local time when it occurred and the values of the two performance counters.

4.2.3 Exception tracing

Some other exceptions can be monitored as well.
The monitored exceptions can be restricted to a list of five thread-ids or to all threads not in the list. Changing the mode will reset the other restrictions. Furthermore monitoring can be restricted to exceptions with an error-code within a specified interval.
The exception number, the fault address, the error-code, the cpu on which

the exception occurred, the local time when it occurred and the values of the two performance counters are displayed.

4.3 Thread names

Threads are identified by thread-ids. These thread-ids are 32 bit hex values. To make it easier for users to recognize specific threads, they can give a thread a nickname. The debugger can hold up to 32 nicknames in the name table.
When a nickname has been associated with a thread, the name is printed instead of the thread-id whenever it occurs.
Some well-known names like *sigma0* can be set in the configuration menu of the debugger (see appendix A.6).

4.4 Data-types

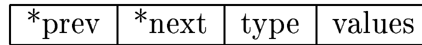
There are two major data-types in the new debugger. The first data-type is a buffer to trace the occurring events to be able to look at them later on. The second one is the name-table where the debugger can store nicknames for the threads.

4.4.1 The name table

The name table is a list of 32 (thread-id, name) pairs. A name can be up to eight characters long.
The name table is written circularly, so when entering the 33rd name, the first one will be replaced.
Entering a new name for a thread whose thread-id is still in the list will rename the thread.

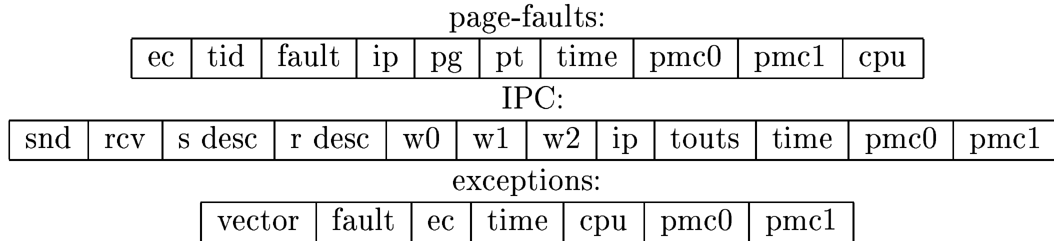
4.4.2 The trace buffer

The trace-buffer is a doubly linked list of type trace_element. There are ten pages of memory reserved for the trace buffer. This equals 718 elements.
The structure trace_element contains a prev and a next pointer and a character defining the type of the traced event. Within the trace buffer there are three structs. One to store the values of traced IPCs, another one to store the values of traced page-faults and a third one to store the values of traced exceptions. To simplify the access to these structures a dword_t array called raw can be used instead. This array consists of twelve dwords. The three



*prev	*next	type	values

Figure 4.1: Structure of trace_element. The field values can either be one of the structures to store information about IPC, page-faults or exceptions or the raw information.



page-faults:

ec	tid	fault	ip	pg	pt	time	pmc0	pmc1	cpu

IPC:

snd	rcv	s desc	r desc	w0	w1	w2	ip	touts	time	pmc0	pmc1

exceptions:

vector	fault	ec	time	cpu	pmc0	pmc1

Figure 4.2: The structure to store traced page-faults, IPCs and exceptions

structures and the raw array are combined in a union. The type field of the structure decides which is used.
When some event has to be traced, the corresponding information is written into the trace buffer.

4.4.3 Restrict events

The events the debugger should display can be restricted. Therefore the debugger contains three records. One for each type of event to be traced (IPC, page-faults and exceptions).
It is implemented as a union of an array for five thread-ids to be restricted and the lower and upper bound of an interval, or an array for five thread-ids to be excluded of the trace.

4.4.4 Trace controlling

To control the behavior of the trace buffer the structure trace_controlling is used. It consists of three major parts. One to control the behavior of traced IPCs, one for traced page-faults and one for traced exceptions. These parts are bit fields of different length.
The IPC field contains four bits. The use bit tells if IPC tracing is turned on or off. The to_do bit decides whether the debugger should be entered after the IPC has been invoked, the restr bit tells the debugger if the threads with thread-id inside or outside the list are restricted. The bit only_sendpart restricts only to IPCs containing a send part.

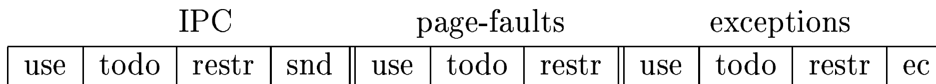


Figure 4.3: Bit-field trace_controlling

The page-fault field contains only three bits. The use bit determines if page-fault tracing is turned on or off, the bit to_do if the debugger should be entered after the page-fault occurs or not and the restr bit, if the threads with thread-id inside or outside the list are restricted.
The exception field contains four bits. Like above the bits use, to_do and restr determine if exception tracing is turned on or off, if the debugger should be entered and how to handle the list. The bit with_errcode determines if the traced exceptions are restricted to ones with a special error-code or not. The aliases YES, NO, TRACING, DISPLAY, THREADS_IN_LIST and ALL_OTHER_THREADS can be used instead of 0 and 1 to make the program easier to be understood. YES and NO stands for use or do not use the feature, DISPLAY means enter the debugger and display the event immediately instead of TRACING. THREADS_IN_LIST and ALL_OTHER_THREADS determine if the threads with thread-id inside or outside the list are restricted. The restrictions are set by default to no tracing, enter the debugger and display the event, restrict to the threads with thread-id in the list and do not restrict to send part or error code.

4.5 Trace buffer dump

To display the contents of the trace buffer the traced events are checked if they match the chosen event type beginning at the latest entry. Then it is checked if the traced event matches the restrictions. If that is true, the event is displayed. The debugger will display up to 718 events on the screen. It is possible to navigate through the buffer by pressing 'n' (next) and 'p' (previous). Pressing 'q' causes the debugger to return to the main menu.

4.6 Implementation of event restriction

The events the debugger should display can be restricted. To check if a event matches the restrictions, the thread-id of the thread causing the event and if necessary the error-code is compared to the entries stored in the corresponding restriction structure.

To differentiate between the possible restrictions, the structure trace_controlling is used (see appendix 4.4.3).

4.7 Configuration via Xconfig

Since the debugger is separated from the kernel many of the debug settings in xconfig are no longer necessary.
To enable the new debugger *L4/KA Kernel Debugger* and *Use new KD* must be switched on. These settings deactivate all the debug code in the kernel sources and cause the new debugger to be linked to the kernel binary.
The next entries choose the debugger's input and output devices. Serial input and output device means that the input is read from and the output is written to the serial port which can be configured. The default value is the first port (COM1 in DOS notation) with 115200 kbit/s, 8N1.
Choosing *kdb* as input device reads the input from the keyboard. Only us-layout is implemented yet. Choosing *screen* as output device the output is written on the screen.
enable disassembler enables the x86-disassembler for the debugger. Its code is copied from binutil's libopcodes and enlarges the debugger by about 60k.
enable debugging, remote enter-KDB and *enter KDB on start* specify the basic behavior of the debugger.

All entries in the trace settings menu have no effect on the new debugger but enlarge the kernel. For this reason they can all be turned off.

The functionality of the debugger depends on the chosen architecture. It is implemented for PIII and it will work on PII too. For any other x86 processor it should be configured as i586.

4.8 Shared files between the old and the new debugger

It is possible to choose either the old or the new version of the debugger while defining the settings of the kernel and the debugger with xconfig. Therefore it is possible to use some functions in both versions.
The x86 files, the files init.c, input.c, mdb.c, print.c, tracepoints.c, kdebug.h and kdebug_keys.h contain functions used by the old and the new version of the kernel-debugger. The main part of the old version is within the file mini-kd.c. The main part of the new debugger is within the file new-kd.c.

Chapter 5

Summary and future work

5.1 Summary

This kernel debugger tries to combine the advantages of the existing debuggers for the L^4 version X debuggers without their disadvantages. The big disadvantage of Hazelnut's debugger is that it is involved in the kernel. The idea to completely separate and hide it from the kernel was coined by the assembler version of the debugger. The solution to this problem is using daisy chaining for debugging the system-calls. To achieve architecture independency is a basic idea of the Hazelnut kernel and it is desirable for the debugger as well. Therefore the debugger is mostly written in C++ and the code is separated into two parts. One part consists of the architecture independent functions of the debugger like the debugging of the kernel data-types. The second part contains the architecture specific functions. By implementing it carefully this makes the debugger also easier to be expanded and maintained. A problem of all old debugger versions is their confusing menus. When starting to work with the kernel it takes a lot of time to learn how to use the debugger. To make its usage easier the functions of the debugger have been arranged in sub-menus containing only one topic.

5.2 Future work

The next steps that have to be taken are implementing the debugger for other platforms, mainly for IA64, and an adaptation to the L^4 Version 4–X.2 API (the L4Ka Pistachio kernel). A GUI for the debugger (e.g. with QT) would also be a very nice item.

Appendix A

Kernel debugger manual

This appendix describes the functionality and the usage of the debugger. The functions in the lists can be invoked by typing the parenthetical symbols.
The hierarchical menus should make the usage of the debugger easier for people beginning to work with it. But there are some shortcuts in every menu to get some frequently used functions faster. A list of these shortcuts is shown in B.2.
An online documentation of the debugger can be found on [7].

A.1 Xconfig

To configure the kernel and the debugger the xconfig menu, a slightly modified Linux TCL/TK configuration utility, can be used by typing *make xconfig* in a shell.
There the functionality and the behavior of the kernel and the debugger can be set. For the new debugger there are only relevant the IO-devices, *enable disassembler, enable debugging, remote enter-KDB* and *enter KDB on start.* All other functions are not supplied or automatically integrated into the debugger. To improve performance and avoid the second print of the information these other functions are turned off, primarily the *trace settings.*

A.2 The main menu

The main menu contains the functions to control the debugger like:

- Reset [ˆ , 6]
- Continue [g]

- Display trace buffer [T]
- Step instruction [s]
- Step block [S]
- The sub-menus [o, c, k, m]

The entries step instruction and step block belong in fact to the CPU sub-menu, but for implementation reasons they have been placed here. The entry point to the debugger should be the same in the old and the new versions of the debugger. But to put the step instruction and step block into the CPU sub-menu, the entry point has to be changed.

A.2.1 Reset

This resets the Computer. The user is asked if it should really restart. Answering yes will immediately reboot the computer. This can cause data loss.

A.2.2 Continue

Returns from the debugger to the kernel/user program.

A.2.3 Display trace buffer

The trace buffer holds a list of 718 events the debugger was told to trace. These could be IPCs, page-faults or exceptions. To trace those events the corresponding trace mechanism has to be turned on in the debugger (see IPC tracing 4.2.1, page-fault-tracing 4.2.2 and exception tracing 4.2.3).
When displaying the trace buffer, the three kinds of entries or any combination of them can be shown. The same information as when displaying the event on occurrence is shown.

A.2.4 Step instruction

This leaves the kernel debugger and executes the next instruction. After that the debugger is invoked immediately. The current stack frame is shown and, if disassembler is enabled, the next instruction is shown too.

A.2.5 Step block

This function is only available on i686 processors. It leaves the debugger and executes all instructions until the next branch instruction is taken. Then the debugger is invoked and the stack frame, the current instruction and the instruction where the branch occurred are displayed.

A.2.6 The sub-menus

There are four sub-menus:

- Configure menu [o]
- CPU menu [c]
- Kernel menu [k]
- Memory menu [m]

These menus are described below. A short description of the sub menus is displayed by pressing '?' ore 'h'.

A.3 The CPU menu

To this menu belong all the architecture dependent parts (except the step instruction and the stop block function). It contains the following sub-menus and commands:

- CPU state [A]
- Performance counters [e]
- The disassembler [U]
- Breakpoints [b]

Most of these functions work on Intel processors only, some of them only on machines with a local APIC like PII/PIII.

A.3.1 CPU state

This menu shows some information of the current state of the CPU. The general descriptor table GDT, interrupt descriptor table IDT, control registers cr0 to cr4, IO-APIC redirection table, the CPU-id and some information for small spaces and IO ports are shown. More information about the tables and registers are described at [4] and [5].

A.3.2 Performance counters

In this menu the user can read and write the two performance counters and assign an event to a specific counter. A list of the possible events can be shown. The possible events are listed in [5], appendix A.1. On machines with local APIC the performance counter overflow handler can be enabled and disabled. When enabled, the debugger is invoked when one of the counters overruns and displays the value of the performance counters (pmcs) and enters the debugger.

A.3.3 Disassembler

When the disassembler is enabled, the address to start disassembling can be chosen. The default setting is the current instruction. By pressing 'u' another instruction can be chosen, 'q' returns to the debugger. Any different key causes the next instruction to be disassembled.

A.3.4 Breakpoints

This function handles breakpoints. The four breakpoint registers of the processor can be set or cleared. A breakpoint can be set to a given instruction 'i', an I/O operation 'o', a memory access 'a' or a memory write access 'w' on the given address. Then the debugger is entered every time the chosen breakpoint is reached.
'+' enables the breakpoint using the same settings as last time this breakpoint was enabled and '-' disables all breakpoints. A list of all breakpoints can be shown with '?'.
Breakpoints that has been set are enabled for all address-spaces. Enabling a breakpoint at address x in address-space A will also enable a breakpoint at address x in address-space B.

A.4 The kernel menu

This menu sumes up the functions to handle all the kernel data-structures and events. Its entries are:

- Kernel-data [#]
- Priorities (of the threads) [Q]
- IPC tracing [i]

- Exception tracing [x]
- Irq assignment [I]
- TCB dump [t]
- Task dump [k]

A.4.1 Kernel-data

This function prints either some statistics about the MDB 'm' or about the kernel memory allocator 'k'.
The mapping database statistics show the depths of the current mapping tree and the number of mappings from a single page frame.
The kernel memory allocator statistics show the amount of free kernel memory and the fragmentation of the free list in the allocator.

A.4.2 Priorities

This function shows the priority queues. If a thread ID is set inside parentheses it indicates that the thread is not in the ready-queue (i.e., not currently runnable).

A.4.3 IPC tracing

Here IPC tracing can be turned on or off.
Pressing a '+' the debugger will be entered before every IPC matching the restrictions and displays the sender's and the recipient's thread-id, the send- and receive-descriptors, the three register words, the timeouts and the performance counters and a timestamp.
Pressing a '*' enters the same values into the trace buffer without displaying them and entering the debugger.
A '-' turns off IPC tracing.

The restrictions can be managed by pressing 'r'. It is possible to restrict the monitored IPCs to a specific thread 't' or to all threads except this thread 'T'. Then the thread-id or the nickname of the thread must be entered. Up to five thread-ids can be specified. Switching between 't' and 'T' causes a reset of the restrictions.
It is also possible to restrict monitoring to IPCs to all threads whose thread-ids fit or don't fit into a specific interval 'x'. Then the boundaries of the interval have to be entered. When the first value is smaller than the second

one, all threads with thread-ids in the interval are monitored. If the first value is larger than the second one, all threads outside the interval are monitored. Equality of both values is not permitted.
The restrictions are reset by pressing '-'.

A.4.4 Exception tracing

Here exception tracing can be turned on or off.
After a '+' all exceptions matching the restrictions are monitored before they are handled and the debugger is entered. The exception, the fault address, the error-code and the performance counters and a timestamp are displayed.
A '*' enters the same information into the trace buffer without displaying and entering the debugger.
Exception tracing is turned off by pressing '-'.

Monitored exceptions can be restricted to up to five specific threads 't' or to all except of them 'T'. Switching between 't' and 'T' causes a reset of the restrictions.
The monitoring can also be restricted to an error-code in a specific interval. The boundaries of the interval have to be entered. When the first value is smaller than the second one, all exceptions with error-code in the interval are monitored. Otherwise all exceptions with error-code outside the interval are monitored. Equality of both values is not permitted.
The restrictions are reset by pressing '-'.

A.4.5 Interrupt assignment

This function shows a list of hardware interrupts and their associated threads.

A.4.6 TCB dump

This function dumps thread control blocks (TCB). The TCB to be dumped has to be entered, thread-ids and TCB addresses are valid inputs. If nothing is specified the current TCB will be dumped.

A.4.7 Task dump

This function displays a list of all threads within the task and a short summary of their current state. The desired task can be chosen by entering a thread-id within the task, a TCB address or a task number. If no task is specified, the information about the current task is shown.

A.5 The memory menu

All functions operating on the memory are in this menu like:

- Memory-dump in the current address-space [d]
- Memory-dump in another address-space [D]
- Page-table dump [p]
- page-fault tracing [f]
- Display the mapping [m]
- MDB tracing [M]

A.5.1 Memory-dump

This function displays 256 bytes of memory starting at a specified virtual memory address in the current address-space. The memory content is displayed as 32 bit hex values and as printable ASCII character. If the dumped memory region is unaccessible #-signs are printed instead of the unaccessible memory contents.
By pressing 'n' (next) or 'p' (previous) the display window will be moved through the virtual memory in the respective direction. The memory dump is closed by pressing 'q'.

A.5.2 Memory-dump in other address-space

It is possible to dump memory in other address-spaces than the current one. The output of this function is the same as in A.5.1, but the address-space to dump is requested. Thread-ids, TCB addresses, task numbers and page table pointers are valid input values. The default value is the current address-space.

A.5.3 Page table dump

This function displays page tables. Thread-ids, TCB pointers and task numbers are valid inputs. If the input does not indicate a valid thread or task, a pointer to a page table is assumed. The default is the current page table.

A.5.4 Page-fault tracing

This function turns page-fault tracing on or off.
Pressing a '+' the debugger will be entered before every page-fault matching all restrictions is handled by the corresponding page-fault handler. The occurrence of the page-fault (user-mode or kernel-mode) is displayed, the thread causing the page-fault, the fault address, the thread's instruction pointer, its pager, the error-code and the content of control register cr3. Furthermore the values of the performance counters and a timestamp are printed.
Pressing a '*' will cause the debugger to enter these values into the trace buffer without entering the debugger and printing the information on the screen.
'-' turns the page-fault tracing off.

The restrictions can be managed by pressing 'r'. It is possible to restrict the monitored page-faults to a specific thread 't' or to all threads except this thread 'T'. A thread-id or the nickname of a thread is requested. Up to five thread-ids can be specified. Switching between 't' and 'T' causes a reset of the restrictions.
It is also possible to restrict page-faults to all threads whose thread-ids fit or don't fit into a specific interval 'x'. Then the boundaries of the interval have to be entered. When the first value is smaller than the second one, all threads with thread-ids in the interval are monitored. When the first value is larger than the second one, all threads outside the interval are monitored. Equality of both values is not permitted.
All restrictions are reset by pressing '-'.

A.5.5 Dump mapping DB

A physical address is requested and the mappings of all virtual page frames containing this address are dumped. The mapping database dump will show a tree like structure describing the mappings between different address spaces and the virtual addresses within the different spaces.

A.5.6 MDB tracing

To use this function MDB tracing has to be enabled via Xconfig.
A '+' turns on the mapping database tracing. Some information about the mapper, the mappee and the mapped fpage are displayed. The debugger is entered after a page-fault has occurred.
A '*' turns on mapping database tracing without entering the debugger after

the occurrence of page-faults.
Pressing '-' turns off mapping database tracing.

A.6 The configuration menu

In this menu it is possible to configure the behavior of the debugger. It contains:

- Name a thread [s]
- Nicknames for some well known threads [n]
- Show the names of the threads [K]
- Enable/disable cache [c]

A.6.1 Name a thread

To make it easier to recognize threads, nicknames can be assigned to thread-ids. A thread-id and a name are requested. Such a name can contain up to eight characters. The debugger prints this nickname instead of the thread-id.

A.6.2 Nicknames for some well known threads

In the L^4 kernel there are some predefined thread-id's like *sigma0, nil* and *invalid*, also the *root task* and the *idler*. This function registers these nicknames.

A.6.3 Enable/disable cache

This function turns on or off the cache while in debugger. The effect of this is described in section 3.5. The cache is turned off by default.

Appendix B

Key-bindings

B.1 Hierarchically ordered Key-bindings

Main menu	
g	continue
ˆ , 6	restart the computer
o	configuration menu
␣	dump frame
m	memory menu
c	CPU menu
k	kernel menu
s	step Instruction
T	dump Trace
S	step Block
?, h	help

CPU menu	
A	CPU state
e	performance counter
U	disassembler
c	profiling
C	dump profile
b	set/clear breakpoints

Kernel menu	
#	dump statistics
Q	list Priority Queues
i	IPC tracing
I	interrupt Association
k	dump Task
t	dump TCB
r	set/clear Tracepoints

Memory menu	
d	dump memory
D	dump memory in other space
m	dump mapping DB
M	trace mapping DB
p	dump page-table
f, P	page-fault tracing

Configuration menu	
c	enable/disable cache while in debugger (Starting disabled)
K	list of nicknames
n	set some known names
s	name a thread

B.2 Shortcuts

The shortcuts are avaliable in every menu to fasten the access to frequently used functions.

Shortcuts for faster use	
b	set/clear breakpoints
T	dump Trace
␣	dump frame
g	continue
?, h	help
t	dump TCB
d	dump memory
D	dump memory in other space
p	dump page-table
ˆ , 6	restart the computer

Bibliography

[1] J. Liedtke: L^4 *Nucleus Version X Reference Manual* (1999)

[2] J. Liedtke: L^4 *eXperimental Kernel Reference Manual*, Version 4-X.2 (May 1, 2001)

[3] J. Liedtke: *LN Kdebug Manual* (1997)

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

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

[6] *L4Ka Home-page*: http://www.l4ka.org

[7] *Online Documentation*: http://i30www.ira.uka.de/~swagner

[8] lecture *Construction of Micro-kernels*, J. Liedtke University of Karlsruhe (2001)

Index

Symbols
$\sigma0$ 13

A
address-space 11, 12
arch_entry 21
arch_exit 22

B
Breakpoints 21, 34

C
cache 21, 39
call 12
configuration 28
configuration menu 39
  cache 39
  name 39
  Nicknames 39
Continue 32
CPU menu 33
  Breakpoints 34
  Disassembler 34
  Performance counters 34
  state 33
cr0 21

D
data-types 25
debug exception 21
disable cache 39
Disassembler 34

E
enable cache 39
Exception 36
exception 24, 25
Exception tracing 24, 36
exception tracing 25

G
grant 13

I
IDT 18
interrupt 1 21
interrupt 14H 24
interrupt 2 21
interrupt 3 21
interrupt 30H 24
Interrupt assignment 36
IPC 11, 12, 24, 25, 35
IPC tracing 24, 25, 35

K
kernel menu 34
  Exception tracing 36
  IPC tracing 35
  IRQ assignment 36
  Kernel-data 35
  Priorities 35
  Task dump 36
  TCB dump 36
Kernel-data 35
key-bindings 41

M
main menu 31
manual 31
map 13

MDB . . . 38
MDB tracing . . . 38
memory dump . . . 37
memory menu . . . 37
    dump MDB . . . 38
    MDB tracing . . . 38
    memory-dump . . . 37
    page table dump . . . 37
    PF tracing . . . 38
memory-dump . . . 37

N
Name . . . 39
name table . . . 25
name_table . . . 23
Nicknames . . . 39
NMI . . . 21

P
page table . . . 37
page table dump . . . 37
Page-Fault . . . 38
page-fault . . . 24, 25
Page-fault tracing . . . 38
Performance counters . . . 34
performance counters . . . 21
PF . . . 24
PF tracing . . . 24, 25
pmc . . . 21
Priorities . . . 35

R
receive . . . 12
receive descriptor . . . 24
Reset . . . 32
restrictions . . . 26, 27

S
send . . . 12
send and receive . . . 12
send descriptor . . . 24
stack . . . 22
Step block . . . 33
Step instruction . . . 32
sub-menus . . . 33
    configuration menu . . . 39
    CPU menu . . . 33
    kernel menu . . . 34
    memory menu . . . 37
syscall
    IPC . . . 25
    page-fault . . . 25
sysenter . . . 24
system-call
    grant . . . 13
    map . . . 13
    unmap . . . 13

T
Task dump . . . 36
TCB . . . 12, 36
TCB dump . . . 36
thread . . . 12, 39
thread names . . . 25
timeout . . . 12
trace buffer . . . 25, 32
trace controlling . . . 26
trace_buffer . . . 23
trace_element . . . 25

U
unmap . . . 13

X
xconfig . . . 14, 28, 31