Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme

Lehrstuhl Systemarchitektur

Porting L4Ka::Pistachio to Mips32

Thomas Blattmann

Studienarbeit

Verantwortlicher Betreuer:	Prof. Dr. Frank Bellosa
Betreuende Mitarbeiter:	Dipl.-Inf. Uwe Dannowski

31. Januar 2006

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 31. Januar 2006

Thomas Blattmann

Contents

1 Introduction

A microkernel is a type of kernel which provides a very simple abstraction over the hardware. Only a minimal set of security relevant services such as thread management, address spaces and interprocess communication are implemented. Other services that are commonly offered by monolithic kernels, including networking, device drivers and pagers are implemented in user-space programs and are not part of the kernel itself. Microkernels aim at extracting as much policy out of the kernel as possible in order to make them suitable for different fields of applications and to keep them easily maintainable.

The L4Ka::Pistachio is the latest L4 Microkernel developed by the System Architecture Group at the University of Karlsruhe. The L4 kernel was originally designed and implemented by Jochen Liedtke (see [4] and [5]) and has become popular due to its outstanding IPC performance. Most architecture independent parts are written in C++ and ports for altogether eight architectures exist.

This work describes the major design decisions that were made when porting L4 to the Mips32 architecture. It also gives some illustrative code extractions. The last two chapters show the result of an IPC performance test and describe implementation alternatives.

The existing Mips64 port by Carl van Schaik served as a template for most design decisions. Some parts were simply adopted and adjusted accordingly. In general, it was tried to keep things as simple as possible while trying to get the most basic things run. Later optimizations (as e.g. described in Section 6) can still be done but were not part of this thesis.

The port made is for a Mips32-4Kc single processor system to which I'll therefore confine myself in the remainder of this document unless it is noted otherwise.

2 Background

This section gives a brief overview of the Mips32-4K architecture, including the Mips32-4Kc processor for which the port was made. Further Simics, a system level instruction set simulator on which all testing and development was done, is introduced.

2.1 The Mips32-4K Processor Core family

The Mips32-4K processor cores are high-performance, low-power, 32-bit RISC cores intended for custom system-on-chip applications. Although the cores implement a 32-bit architecture, many features (e.g. the MMU) are modeled after the ones that can be found in the 64-bit R4000 family. The 4K family has three members. The 4Kc contains a fully associative Translation Lookaside Buffer (TLB) based MMU and a pipelined multiply-divide unit (MDU). The TLB itself consists of three address translation buffers: a 16 dual-entry, fully associative Joint TLB (JTLB), a 3-entry instruction micro TLB (ITLB), and a

3-entry data micro TLB (DTLB). Both micro TLBs are hardware managed and inaccessible to the system programmer. The JTLB, however, is controlled and refilled solely by system software.

The 4Km and 4Kp processors use a fixed mapping mechanism instead of a TLB-based MMU, which performs a simple translation to get the physical from the virtual address. Unmapped segments are treated identically by all cores and are described in [2], Chapter 3. The simple translation takes part in the lower 2 GB of the virtual address space. For more details refer to [1], Chapter 3, "Memory Management". While the 4Km processor uses a pipelined MDU as in the 4Kc core, the 4Kp processor has a smaller non-pipelined MDU.

Optional data and instruction caches can be flexibly configured at processor built time for various sizes, organizations and set-associativities. A System Control Coprocessor (CP0) is responsible for virtual-to-physical address translation, cache protocols, the exception control system, operating mode selection, and the enabling/disabling of interrupts. It also provides configuration information such as cache size, TLB size and associativity. CP0 is part of each Mips CPU even though its name might suggest that it is not.

All cores execute the Mips32 instruction set architecture (ISA) containing all Mips II instructions as well as special multiply-accumulate, conditional move, prefetch, wait or zero/one detect instructions.

2.2 Register Set

The processor has 32 user accessible general-purpose registers. The hardware itself makes few rules, but conventions on the usage of the different registers exist. The register called *zero* is hard wired to zero. *at* is reserved for the assembler to translate pseudo instructions into actual instructions or sequences of instructions. It is commonly not used by the programmer. *v0* and *v1* are used to return values from functions. *a0* to *a3* contain the first four parameters that are passed to a function and pointers to them, respectively. *t0* to *t9* are for general purpose. They are not preserved through function calls. *s0* to *s8* are to be saved and restored by the callee so that the register content is the same as before when returning from a function call. *fp* is intended to be a frame pointer, *gp* a global pointer which the compiler can use to point into a static memory region for a faster (gp-relative) access. *sp* holds the current stack pointer and *ra* the return address from a function call. *k0* and *k1* are both reserved for the operating system. Their content can change any time to arbitrary values from a user application's point of view.

Coprocessor0 has 32 registers to be used by the operating system. That register set is only accessible in kernel mode.

General purpose registers are referred to as GP[name] hereafter. CP0[name/bits] denotes a Coprocessor0 register in which the optional 'bits' names a bitfield within a register.

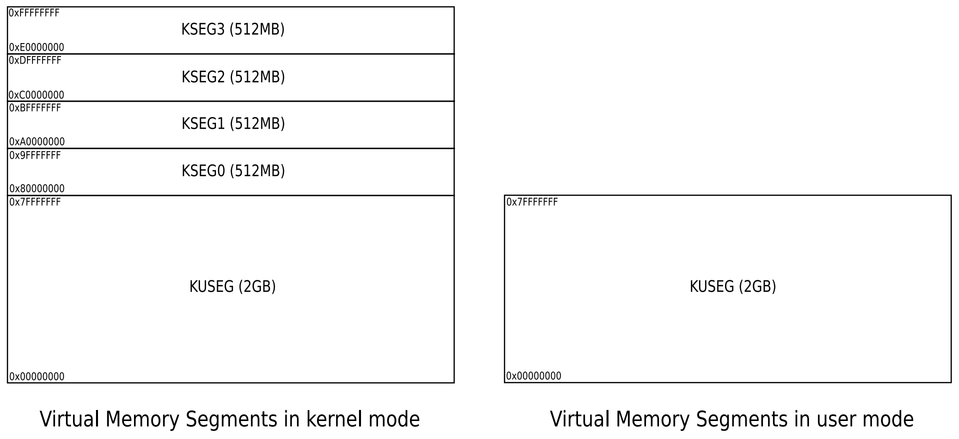

Figure 1: Virtual Memory Segments

2.3 Memory Management

2.3.1 Virtual Memory Segments

The 4 Gigabyte address space is divided into five areas as can be seen in Figure 1.

kuseg The low 2 Gigabyte, denoted kuseg, are addresses permitted in user mode and kernel mode. The system maps all user mode references to kuseg through the TLB. Data and instructions in this segment can be cached once the cache is initialized. In kernel mode, the processor can be set up to map kuseg's virtual addresses directly to the same physical addresses, or to use the TLB for translation.

kseg0 The 512MB of virtual memory in kseg0 are translated contiguously into the low 512MB of physical memory. This region is accessed through the cache. The main parts of the kernel are commonly loaded and executed in this segment.

kseg1 The next 512MB make up kseg1. They give a duplicate mapping of the first 512MB of physical memory. Accessing kseg1 will not involve the cache and is thus the only chunk of memory map that is guaranteed to behave properly from system reset. It is therefore used to access the initial program ROM.

kseg2/kseg3 Being in kernel mode, software has access to the entire address space including kseg2 and kseg3 with 512MB each. They are both translated through the TLB. The mapping of kseg3 changes if Debug mode is entered. In Debug mode, a special debug segment is mapped into kseg3.

2.3.2 Translation Lookaside Buffer

The 4Kc core implements a 16 dual-entry, fully associative Joint TLB that maps 32 virtual pages. It is organized as pairs of even and odd entries to minimize the overall size. Each pair is tagged by an 8-bit Address Space Identifier (ASID) and translates two consecutive pages that can range from 4KB to 16MB. The ASID is compared to CP0[entryhi/asid] each time a translation proceeds. Entries whose ASID tag does not match CP0[entryhi/asid] are ignored. A global bit can be set for each double entry. If set, the ASID field is ignored on lookups and the pages are global to all address spaces.

Apart from a CP0 register that holds the current ASID, the TLB makes up the whole MMU hardware. The Operating System uses the TLB as a cache to a memory-resident page table. Thus, no particular format of page table is needed. When presented with an address it can't translate, the TLB triggers a special exception to invoke a software routine. It is then up to system software to put the faulting address into the TLB or to handle the exception appropriately.

2.3.3 Cache

A 4Kc processor supports separate instruction and data caches, which allows instruction and data references to proceed simultaneously. Both caches are virtually indexed and physically tagged. This way, cache access and virtual-to-physical address translation can be done in parallel. As the physical address is used to tag each cache line, flushing the cache on a context switch should not be necessary in most cases. Each cache line is completed with valid bits for each data word cached as well as a locked bit to prevent cached data from getting replaced.

The write-through strategy eases things with respect to cache programming. Once the cache is initialized only little management effort should be necessary. The programmer has to be aware of possible timing delays for I/O register access caused by a write buffer. Further, the system software has to handle cache coherency issues within the memory hierarchy that can affect an additional memory master (e.g. a DMA controller). The OS's memory management must also avoid cache aliases that can arise when the same physical address may be described by different virtual addresses of different threads.

2.3.4 Exception Handling

The processors receive exceptions from a number of sources, including TLB misses, arithmetic overflows, I/O interrupts, and system calls. Whenever the CPU detects an exception, the normal sequence of instructions is suspended and the processor enters kernel mode, disables interrupts and jumps to a software exception handler. Table 1 shows the most common exceptions along with their corresponding vectors. For a detailed description refer to [2] and [3].

On an exception, CP0[epc] contains the instruction pointer (IP) that points to the instruction where execution should be resumed after the exception. CP0[cause] is set up so that software can see the reason for the exception. On a TLB-miss

EXCEPTION	VECTOR
Reset, Soft Reset, NMI	0xbfc00000
TLB miss exception	0x80000000
TLB invalid exception	0x80000180
Interrupts	0x80000180
Breakpoints	0x80000180
All other exceptions	0x80000180

Table 1: Exception Vectors

exception CP0[badvaddr] contains the faulting address. Other than that no registers are saved by hardware.

Mips exceptions are precise, meaning that all instructions preceding the exception victim in instruction sequence are completed; any work done on the victim and on any subsequent instructions (due to the pipeline) has no side effects the software needs to worry about.

2.4 Simics Simulator

Simics is a system level instruction set simulator supporting various target systems. It models the Mips32-4Kc processor with a limited set of devices from the Mips Malta development board. All development and testing has been done on version 2.2.19. Earlier versions contain bugs (e.g. movez instruction) and prevent the kernel from executing properly.

Unfortunately, there are a few limitations and features not yet supported by Simics/Mips. I state the ones with an immediate impact on what was implemented (confer to Section 4.4, Restrictions). Other limitations can be found in [6].

- There are a few instructions that have not yet been implemented. Other instructions are simply 'nops' and doing nothing. One of them is the 'cache' instruction, which is used for cache management. Another one is 'wait', used to enter standby mode.
- As no cache model has been connected to Simics/Mips, reads and writes to the corresponding control registers in CP0 have no effect at all.
- The simulator supports little-endian mode only.
- Simics/Mips doesn't consider instruction hazards caused by the pipeline. A real Mips processor would need nops following some instructions to

0x00000000	128MB of RAM are mapped at address zero
0x18000070	Dallas Semiconductor DS12887 real-time clock (CMOS clock)
0x180003f8	National Semiconductor PC16550 serial port controller (UART)
0x1f000000	Malta board specific LCD display
0x18000000	Intel i8259 interrupt controller

Table 2: Memory mapped I/O-devices

keep the pipeline happy. As a consequence, execution on Simics works but might fail on real Mips hardware

- No boot loader is available. Instead the kernel is loaded directly into Simics and no boot information is provided.
- It does not implement a floating-point unit.

As is common on the Mips architecture, all IO devices are memory mapped. Table 2 shows where in physical memory what devices are mapped in the Simics configuration that was used.

3 Major Design Decisions

This section describes the major design decisions made when porting L4 to the Mips32 architecture. It first gives details on memory management, continues with Thread Control Blocks, Resource Management and the exception handling mechanism and is closed with a description of the L4 API. The Mips64 port served as a template for many design decisions. Hence some design was simply adopted and adjusted accordingly.

3.1 Memory Management

3.1.1 Physical Memory Layout

The kernel ELF binary spans around 190KB and is loaded into memory starting at physical address 0x400. The area below contains entry points for exception handlers. The following 16MB are reserved for use by the kernel memory allocator. All remaining physical memory is registered in the KIP and made user accessible.

3.1.2 Virtual Memory Layout

The area from 0x00000000 to 0x80000000 (lower 2GB) is available for user applications. The whole kernel is linked into kseg1. kseg0 remains unused as result of the lacking cache simulation. The lower 8MB of kseg2 are reserved to establish a temporary mapping in an address space (communication window) on a Long IPC (cf. to [4]). The remainder of kseg2 and kseg3 contain the virtual array of Thread Control Blocks (see Section 3.2).

3.1.3 I/O-Ports

The physical memory area containing I/O-devices can be shared among various address spaces. It is put into the KIP with memory descriptors set to 'shared'.

3.1.4 Address Spaces

Each address space contains a page directory and an Address Space Identifier (ASID). The page table is organized hierarchically. One page directory per address space and up to 1024 pages can map up to 4GB virtual address space. For **user address spaces** the page tables need to be able to map the first 2GB and a communication window while kseg0/1 and most parts of kseg2/3 remain unused. Hence, parts of the page table that are not needed are used to store some address space management related data. The **kernel address space** on the other hand only has to map the TCB area. The lower half is unused.

There are no restrictions on how the page tables are organized in memory as the TLB is software loaded on a miss. Nevertheless, data in the page tables are stored so that they can be written into the TLB without further conversions. Page tables are allocated on demand. Each entry has 32bits. The provided Linear Page Table Walker was used for virtual-to-physical address translation as well as FlexPage mapping and unmapping. It was set up accordingly.

The processor uses an Address Space Identifier to keep TLB entries from different address spaces apart. Because the architecture supports only 512 ASIDs, an ASID engine that had already been used in some other L4 ports was taken. Whenever a thread in a certain address space is scheduled, it is checked whether a valid ASID is already assigned. If it's not, an allocated ASID is preempted from another address space according to an LRU algorithm and assigned to the scheduled thread's one after cleaning up the TLB.

3.1.5 Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) uses a simple LIFO replacement strategy as offered by the hardware. User translations are tagged with an Address Space Identifier. Consequently they do not need to be flushed on a context switch. Apart from the TCB area, the kernel's code and data are located in memory segments that do not involve the TLB. TCB area translations are marked global when put into the TLB, implying that TCBs can be accessed in kernel mode independently of CP0[entryhi/asid].

The cacheability flags are set for each TLB entry to indicate whether the data is cacheable or not. Writable and valid bits are used as provided by the hardware. There is no hardware dirty bit to indicate whether a translation was accessed or not, however.

One TLB dual entry is hard wired and set up to provide the translation of the currently running thread's TCB. The reason for this is described in section 3.4.2.

3.1.6 Mapping Database

A mapping database is the kernel's view of all address spaces, i.e. recursively mapped or granted FlexPages. Each mapping entry in the page table is supplied with an additional link to its mapping database entry and vice versa.

The Mapnode links for page table entries are located in an array appended to each user page table. In order to obtain the link, an offset of 4KB needs to be added to the address of the corresponding page table entry.

3.2 Thread Control Blocks

Two segments, kseg2 and kseg3 contain a virtual array of Thread Control Blocks (TCBs) with 4096 Bytes each. The size of this area limits the total number of active threads in the system to 254K. TCBs are mapped only in the kernel's address space. No virtual area is shared among different address spaces, making any kind of synchronization superfluous.

A Thread Control Block contains some thread related data (like local/global id, state, pointer to UTCB) as well as a stack (see [4]) and is only accessible in kernel mode. The stack is used by threads running in kernel mode, i.e. whenever a thread switches from user to kernel mode its stack pointer is set to point to the kernel stack within its TCB. The user context is always stored at the very bottom of the stack.

3.3 Resource Management

The communication area for the temporary mapping during Long IPC is currently the single resource in use as no Floating Point Unit was available. The lower 8MB of the kseg2 segment are reserved to establish that mapping. There is one reserved area per address space. An existing mapping is saved on each Thread Switch and restored when switching back to the thread.

3.4 Exception Handling

Exceptions include interrupts, system calls, TLB misses, breakpoints, invalid operations or privileged operations performed in user mode. On an exception, the processor switches to kernel mode and continues execution at predefined locations within the exception vector area. Interrupts are turned off and the faulting thread's context is stored on the kernel stack in its TCB. A global, kernel accessible variable always holds a pointer to the currently running thread's kernel stack. Interrupts remain turned off in kernel mode and are not re-enabled until it is switched back to user mode or to the idle thread.

The kernel initializes exception vectors at system startup, i.e., exception code is copied to the proper addresses in physical memory. Since there are only 128 bytes available between different vectors, the copied code does nothing but jump to the actual exception handler routines.

Several exceptions share the same exception handler located at physical address 0x00000180. To be able to keep them apart, the exception code field in CP0[Cause] is set by hardware. This value is used as an index into a pointer array with pointers to the respective exception handler.

3.4.1 Interrupts

The architecture distinguishes eight different interrupts sources: Two software and six hardware interrupts. The timer interrupt is used by the kernel and cannot be forwarded to user applications. Software interrupts remain unused. The other five hardware interrupts are only acknowledged by the kernel but not used for internal purposes. User threads can register themselves to get notification whenever one of those interrupts occurs.

On interrupts, the whole user context is stored. The kernel uses a pointer array, indexed by the interrupt number, to jump to the appropriate interrupt handler. This array is initially set up to let handle_interrupt(), which delivers interrupt IPCs, be the default interrupt handler.

The interrupt exception handler checks the bits in CP0[cause] to find out the interrupt source. It starts with the highest bit that is occupied by the timer and continues with the next lower etc. This order defines the priority of each interrupt in cases in which multiple interrupts occur at the same time.

3.4.2 TLB-miss exception

A TLB-miss exception is initiated in user and kernel mode when a TLB-mapped virtual address is referenced and the translation is not available in the TLB. A higher-level C routine is responsible for looking up the faulting address and updates the TLB if a translation is found in the pagetable. Prior to invoking it, the interrupted thread's context is stored on the thread's kernel stack. A page fault (no matching translation found in page table) causes the routine to call the main L4 pagefault handler (space_t::handle_pagefault). On return, the missing mapping is looked up again in the page table and the translation is put in the TLB if available. The thread's context is restored and execution continues with the faulting instruction.

A TLB-miss on the currently running thread's TCB would result in a never-ending TLB-miss recursion. It arises when the kernel tries to store the faulting threads context on the kernel stack in the TCB. This access causes another TLB-miss before the first one could be handled successfully and so on. Therefore, the TCB translation for the current thread is hard-wired in the TLB and updated on a thread switch.

3.4.3 Breakpoints

The 'break' instruction is another way to enter the kernel and to continue execution at a predefined location, respectively. It is mainly used to enter the kernel debugger when one of the breakin functions (for debugging purposes) is invoked. It can also be used by user applications to perform 'putc' and 'getc' I/O functions.

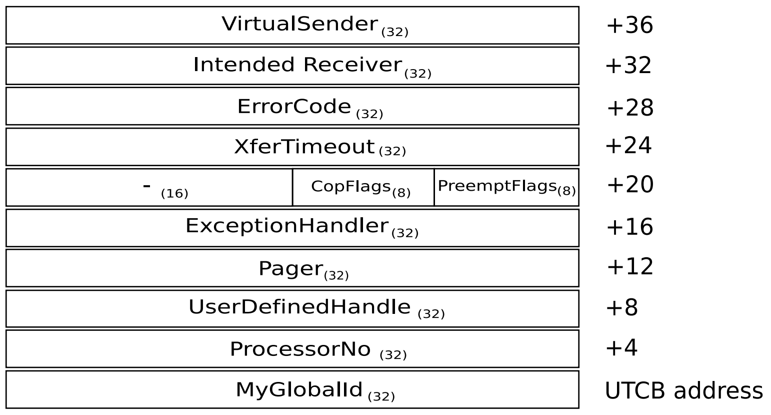

Figure 2: Thread Control Registers

3.5 L4 API

3.5.1 Thread Control Registers

Thread Control Registers (TCRs) are part of the User Thread Control Blocks (UTCB). The address of the current thread's UTCB is the same as the thread's local ID, and is thus immutable. The UTCB (and hence local ID) is made available in the GP[k0] register by the kernel. GP[k0] should be treated as read-only. If modified, the effects are undefined.

Message registers MR[0] through MR[7] map to the processor's general purpose registers GP[s0] to GP[s7] for IPC. The remaining message registers map to memory locations in the UTCB. MR[8] starts at byte offset 96 in the UTCB, and successive message registers follow in memory.

All **buffer registers** map to memory locations in the UTCB. BR[0] is at byte offset 320, BR[1] at byte offset 324, etc.

3.5.2 Exception Messages

The kernel handles some exceptions. A default exception handler exists for all other exceptions. If there is a user exception handler registered with the faulting thread, it generates and sends an exception message as shown in Figure 3. In reply, the handler thread can set the faulting thread's IP, SP and user flags. If there is no handler thread registered, the kernel debugger, which is part of current L4 implementations, is entered.

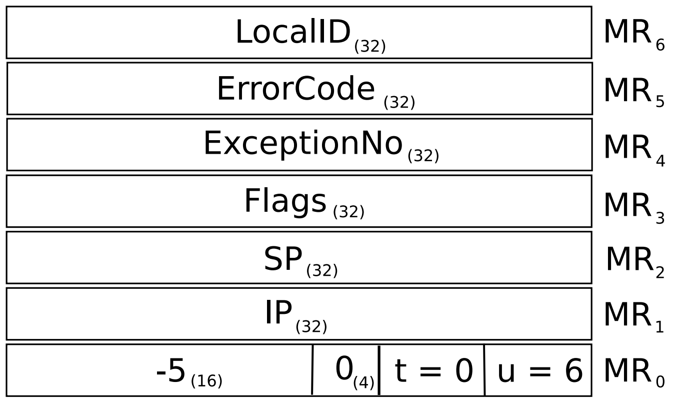

Figure 3: General Exception Message

The following is a table of values for the *ExceptionNo*:

Exception	ExceptionNo	ErrorCode	Delivered
Interrupt	0	-	No
TLB Write Denied	1	-	No
TLB Miss Load	2	-	No
TLB Miss Store	3	-	No
Address Error (load/execute)	4	BadVAddress	Yes
Address Error (store)	5	BadVAddress	Yes
Bus Error (instruction)	6	-	Yes
Bus Error (data)	7	-	Yes
System Call	8	-	v0 $\geq$ 0
Break Point	9	-	!(-104 $\geq$ AT $\geq$ -100)
Reserved Instruction	10	Instruction	AT $\neq$ 0x141fcall
Coprocessor Unavailable	11	Number	CP0, CP2, CP3
Arithmetic Overflow	12	-	Yes
Trap	13	-	Yes
Watch Point	23	-	Unless used by kdb
Machine Check	24	-	Yes

Note, not all of these exceptions will be delivered via exception IPC. Some will be handled by the kernel. Delivered exceptions are indicated in the last column of the table above.

3.5.3 System Calls

The system call entry code is located in the Kernel Interface Page area. Pointers within the KIP contain their absolute addresses. In general, it is tried to follow the o32 calling conventions whenever possible, i.e. parameters are passed in general-purpose registers (GP[a0]-GP[a4]) and results are returned in GP[v0]. This works out for system calls with less than five parameters and a single return value. Other system calls pass additional parameters in GP[s0] to GP[s3] in order not to rely on a valid user stack pointer. Additional return values are returned in GP registers as well. The kernel expects GP[v0] to contain a constant indicating which system call is to be performed.

Generally, registers accessible by user applications are not preserved across system calls but contain return values or are undefined. Appendix A shows for each implemented system call the register content before and after the 'syscall' invocation.

4 Implementation

This sections shows how syscalls and thread switches are done. Both transactions are explained along with code extractions. Finally, restrictions of the current implementation are stated.

4.1 Compiler and ABI

The port was made on an x86 machine using a mipsel-unknown-linux-gnu gcc cross compiler version 3.4.2 with traditional Mips o32 calling conventions. A description of the o32 standard can be found in [1], Chapter 10, "C Programming on Mips".

4.2 System Calls

4.2.1 Invocation in user mode

A user application is provided a library that contains the actual kernel entry code. Prior to executing the syscall instruction, parameters and the syscall identifier constant are put into predefined GP registers. The kernel expects a valid syscall identifier in GP[v0] and parameters in GP[a0]-GP[a4] as well as GP[s0]-GP[s3] when needed. The following code illustrates what is done in user mode to perform a L4_ThreadControl system call. It is assumed that all parameters have already been loaded into GP registers.

```
__asm__ __volatile__ (
        "lw $2, __L4_ThreadControl  \n\t"  /* entry point in KIP (absolute) */
        "subu $29, $29, 0x10        \n\t"  /* space on the stack as by o32 */
        "jalr $2                    \n\t"  /* jump */
        "addu $29, $29, 0x10        \n\t"  /* clean up stack */
        "move %0, $2                \n\t"  /* result */
        : "=r"(r_result)
        : "r"(r_dest), "r"(r_space), "r"(r_schedule), "r"(r_pager), "r"(r_utcb)
);
```

The code located in the KIP area sets GP[v0] and performs the 'syscall' instruction:

```
    BEGIN_PROC(user_thread_control);
       li   v0, SYSCALL_thread_control;     # syscall identifier
       syscall;                             # kernel entry
       j    ra;
    END_PROC(user_thread_control);
```

The 'L4_KernelInterface' system call is an exception to the general proceeding since the KIP and the 'syscall' instruction are not used. Instead, an illegal instruction is executed to enter kernel mode after a magic value was placed in GP[at]. On an illegal instruction, the kernel compares GP[at] with that magic constant and executes the L4_KernelInterface system call if they match.

4.2.2 Handling in kernel mode

A 'syscall' instruction in user mode causes the processor to switch to kernel mode and to continue execution at a predefined location. Apart from some essential registers (e.g. Instruction Pointer, Stack Pointer) no user context is saved. The syscall handler puts all arguments that were passed in GP[s0]-GP[s3] on the stack as by the o32 calling conventions and jumps to higher level C routines.

```
[...]
     # save some user context on kernel stack
[...]
```

```
     # jump here on return from sys_thread_control
     addiu ra, %lo(mips32_l4syscall_five_in_return)

     subu sp, sp, 0x14  # Space for 5 in parameters
     sw s0, 0x10(sp)    # pass arg no 5 on stack

     j sys_thread_control;
     nop
[...]
```

4.2.3 Return from Syscall

The kernel puts all returned values in GP registers, restores some user context and finally executes the 'eret' instruction in order to continue execution in user mode.

4.3 Context Switch and Notifications

On a Thread Switch, the running thread stores its global pointer, frame pointer and return address on its kernel stack. The stack pointer is stored at a fixed location within the 'old' thread's TCB. The actual switch takes place in two steps after the hard-wired TLB entry was updated. First, the new ASID is put into CP0[entryhi]. Subsequently, the stack pointer is set to its new value. The new thread simply restores its switch context and returns.

The following code extraction illustrates this procedure. Before it is executed, an ASID for the target address spaces is determined and stored in the local variable 'new_asid'.

```
__asm__ __volatile__ (

/* execution of the old thread */

    "addiu  $29,$29,-20       \n\t" /* allocate switch frame */
    "la     $31,0f            \n\t"
    "sw     $16,0($29)        \n\t" /* save registers */
    "sw     $17,4($29)        \n\t"
    "sw     $30,8($29)        \n\t"
    "sw     $28,12($29)       \n\t"
    "sw     $31,16($29)       \n\t"

/* hard wire TCB translation */
[...]


/* the actual thread switch */

    "mtc0   %[new_asid], "STR(CP0_ENTRYHI)" \n\t" /* set new asid */
    "move   $29, %[new_stack]              \n\t" /* set new sp */
    "or     %[new_stack], 4096-1           \n\t"
    "addiu  %[new_stack], 1                \n\t"
    "sw     %[new_stack], 0(%[stack_bot])   \n\t" /* "epc0" */

/* execution of the new thread */

    "lw     $31,16($29)                    \n\t"
    "lw     $16,0($29)                     \n\t"
    "lw     $17,4($29)                     \n\t"
    "lw     $30,8($29)                     \n\t"
    "lw     $28,12($29)                    \n\t"
    "addiu  $29,$29,20                     \n\t"

    "jr     $31                            \n\t"
    "0:                                    \n\t"
);
```

As explained in section 3.4.2, the hard-wired TLB entry is updated on each thread switch. That code part was omitted in the extraction above. It is quite straightforward though: First it is checked, whether a valid translation is already available in the TLB. If not, the target TCB is simply touched to cause a TLB miss. Finally, the old hard-wired entry and the entry to be hard-wired are swapped in order to place the new one into the hard-wired area.

Notify functions enable the kernel to force a thread to execute specific kernel functions when scheduled before it continues execution normally. All such a function has to do now is to put a 20 byte switch stack on top of the interrupted threads kernel stack that contains the function to call, up to two arguments and the return address.

4.4 Restrictions

Even though the implementation runs flawlessly on Simics, there are limitations. The kernel in the current state won't run on real hardware. The main reason is the cache management, which has been ignored. Another one are CP0 hazards that might arise even though nops were added according to [3]. It has never been tested on real hardware though. The missing boot loader (and thus memory

untyped words sent	instructions counted	difference
0	918	-
1	992	+ 74
2	1014	+ 22
3	1036	+ 22
4	1058	+ 22
5	1080	+ 22
62	2552	-
63	2574	+ 22

Table 3: Short IPC (Inter-AS)

descriptors) made it necessary to hard code the entry point address of Sigma0, Sigma1 and the Roottask. As stated above Simics/Mips does not implement the 'wait' instruction that could have been used in the idletask to enter standby mode while waiting for interrupts. Instead, a global variable is set to indicate a timer interrupt. The idletask stays in a loop actively waiting for the variable to become one.

Two syscalls are unimplemented, that is ProcessorControl and MemoryControl. Finally, interrupts are not enabled during a Long IPC, delaying the context switch and potentially allowing the communicating threads an excessive time slice.

5 Evaluation

The Mips32 port described in this work targets the simulated Malta board with a Mips-4Kc processor. Since no real hardware was available, the performance is evaluated in Simics instructions. Table 3 shows the number of instructions that were counted for a Short Inter-AS IPC with different numbers of untyped words and no types words.

6 Conclusion

The L4 microkernel port that is described in this document was done in the context of my study thesis "Porting L4Ka::Pistachio to Mips32". It's primary goal was to get a working port for the Mips32 core and little attention was given to optimizations.

The port was tested and developed on the Simics instruction set simulator (cf. Section 2.4). Test cases comprise an L4 test suite as well as some available user applications (including PingPong and GrabMem). Nevertheless there remain parts of the implementation that were not covered by any test cases. There is no guarantee that those parts and also the tested parts are free of errors.

Having basic knowledge of the L4 internals, a port to a new architecture can be done quite straightforwardly as a result of existing C++ template classes and methods. These templates compose a documented framework of the as yet unimplemented architecture dependent parts of the kernel that are called from somewhere within the architecture independent parts. Only those classes and methods are to be implemented while architecture independent parts can be thought of as a black box that you do not need to worry about. The templates do not impose any restrictions on design and implementation and give the programmer the freedom to create whatever is best suited to the underlying architecture.

The Mips32 port uses those templates and made no changes to architecture independent code necessary. Many design decisions as well as some source code could be easily adopted from the port of the related Mips64 architecture.

Apart from restrictions as stated in Section 4.4, L4/Mips32 works flawlessly when executed on Simics and passes all test cases.

6.1 Suggestions for future work

TLB miss handling could be implemented faster. As is, the whole user context is stored and the translation is looked up and inserted into the TLB in a higher level C routine. If a pointer to the page directory was stored at a fixed address or within an unused CP0 register (e.g. CP0[context/PTEbase]), the faulting address could be looked up and inserted into the TLB within a few cycles. A second, more powerful handler would only be invoked if no translation is found in the page table (pagefault).

What the Mips architects had in mind was a contiguous non-hierarchical virtual 2MB page table in kseg2 for each user address space. This, on the one hand, saves physical memory since the unused gap in the middle of the page table will never be referenced. On the other hand it provides an easy mechanism for remapping a new user page table when changing context. By simply changing the CP0[entryhi/asid] value on a context switch, the pointer to the page table is automatically remapped onto the correct page table. A TLB-miss recursion, caused by TLB-misses within the kseg2 area, can be avoided by having a slower, more general handler routine that gets invoked on nested exceptions. The Mips hardware supports such a kind of linear page table in form of the CP0[context] register. If the virtual page table starts at a 1MB boundary and CP0[entryhi/PTEbase] is set up with the high-order bits of the pagetables base, the Context register will automatically point to the address of the required translation on each user refill exception.

One would need to reserve 2MB in kseg2 for the virtual page table. The global bit for that area would not be set in order to have the page tables per address space. An additional 512 byte per address space within an unmapped section (e.g. kseg0) could be used as page table to translate that 2MB area.

A never-ending TLB-miss recursion as described in section 3.4.2 is avoided by using hard-wired entries. You could also provide an additional static kernel stack to handle nested miss exceptions. The static stack would be global to all threads and used on TLB-miss exceptions that arise when touching a thread's kernel stack.

By using a fast TLB-miss handler (that does not save any context) as described above, one could even omit the hard wired TLB entry if it is guaranteed that the TCB is mapped in the page table on a miss.

References

[1] Dominic Sweetman: *See Mips Run,* Morgan Kaufmann Publishers

[2] Mips32 4K Processor Core Family Software User's Manual Revision 01.17 (2002)

[3] Mips32 Architecture For Programmers Volume III: The Mips32 Privileged Resource Architecture Revision 2.50 (2005)

[4] Jochen Liedtke: *Improving IPC by Kernel Design* 14th ACM Symposium on Operating System Principles (1993)

[5] Jochen Liedtke: *On u-Kernel Construction* 15th ACM Symposium on Operating System Principles

[6] Simics/Mips Target Guide Version 2.2.19 (2005)

A Systemcalls

A.1 KernelInterface

GP[at]	0x141FCA11		[id]
GP[v0,v1]	[ign]		[id]
GP[a0..a3]	[ign]		[id]
GP[t0]	[ign]		KIP base address
GP[t1]	[ign]		API Version
GP[t2]	[ign]	KernelInterface	API Flags
GP[t3]	[ign]	opcode 0xF1000000	Kernel ID
GP[t4..t7]	[ign]		[id]
GP[s0..s7]	[ign]		[id]
GP[t8,t9]	[ign]		[id]
GP[gp,sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[id]

A.2 Schedule

GP[at]	[ign]		[inv]
GP[v0]	[ign]		result
GP[v1]	[ign]		[inv]
GP[a0]	dest		time control
GP[a1]	time control		[inv]
GP[a2]	processor control		[inv]
GP[a3]	priority		[inv]
GP[t0..t9]	[ign]	Schedule	[inv]
GP[s0]	preempted control	$\rightarrow$	[id]
GP[s1..s7]	[ign]		[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.3 ExchangeRegisters

GP[at]	[ign]		[inv]
GP[v0]	[ign]		result
GP[v1]	[ign]		[inv]
GP[a0]	dest		control
GP[a1]	control		SP
GP[a2]	SP		IP
GP[a3]	IP		flags
GP[t0]	[ing]		pager
GP[t1]	[ign]		userhandle
GP[t2..t9]	[ign]	ExchangeRegisters	[inv]
GP[s0]	flags	$\rightarrow$	[id]
GP[s1]	userhandle		[id]
GP[s2]	pager		[id]
GP[s3]	local		[id]
GP[s4-s7]	[ign]		[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.4 SpaceControl

GP[at]	[ign]		[inv]
GP[v0]	[ign]		result
GP[v1]	[ign]		[inv]
GP[a0]	space		control
GP[a1]	control		[inv]
GP[a2]	kip		[inv]
GP[a3]	utcb area		[inv]
GP[t0..t9]	[ign]	SpaceControl	[inv]
GP[s0]	redirector	$\rightarrow$	[id]
GP[s1..s7]	[ign]		[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.5 ThreadSwitch

GP[at]	[ign]		[inv]
GP[v0, v1]	[ign]		[inv]
GP[a0]	dest		[inv]
GP[a1..a3]	[ign]		[inv]
GP[t0..t9]	[ign]	ThreadSwitch	[inv]
GP[s0..s7]	[ign]	$\rightarrow$	[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.6 Clock

GP[at]	[ign]		[inv]
GP[v0]	[ign]		ticks (low 32 bits)
GP[v1]	[ign]		ticks (high 32 bits)
GP[a0..a3]	[ign]		[inv]
GP[t0..t9]	[ign]	Clock	[inv]
GP[s0..s7]	[ign]	$\rightarrow$	[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.7 Unmap

GP[at]	[ign]		[inv]
GP[v0,v1]	[ign]		[inv]
GP[a0]	control		[inv]
GP[a1..a3]	[ign]		[inv]
GP[t0..t9]	[ign]	Unmap	[inv]
GP[s0..s7]	[ign]	$\rightarrow$	[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.8 IPC

GP[at]	[ign]		[inv]
GP[v0]	[ign]		result
GP[v1]	[ign]		[inv]
GP[a0]	to		[inv]
GP[a1]	from		[inv]
GP[a2]	time		[inv]
GP[a3]	[ign]		[inv]
GP[t0..t9]	[ign]	IPC	[inv]
GP[s0]	mr0	$\rightarrow$	mr0
GP[s1]	mr1		mr1
GP[s2]	mr2		mr2
GP[s3]	mr3		mr3
GP[s4]	mr4		mr4
GP[s5]	mr5		mr5
GP[s6]	mr6		mr6
GP[s7]	mr7		mr7
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]

A.9 ThreadControl

GP[at]	[ign]		[inv]
GP[v0]	[ign]		result
GP[v1]	[ign]		[inv]
GP[a0]	dest		[inv]
GP[a1]	space		[inv]
GP[a2]	schedule		[inv]
GP[a3]	pager		[inv]
GP[t0..t9]	[ign]	ThreadControl	[inv]
GP[s0]	utcb	$\rightarrow$	[id]
GP[s1..s7]	[ign]		[id]
GP[gp]	[ign]		[inv]
GP[sp]	[ign]		[id]
GP[s8]	[ign]		[id]
GP[ra]	[ign]		[inv]