— Draft of April 22, 2002 —

Prototypical Design and Implementation of L4-SMP Microkernel Mechanisms

Marcus Völp

Supervisor: Prof. Dr. Jochen Liedtke

Universität Karlsruhe

1 Einleitung

Speichergekoppelte Multiprozessorrechner finden ihren Einsatz als mittelgroße Serverrechner und Hochleistungs- Arbeitsplatzrechner. Symmetrische MultiProzessor (SMP) Systeme bilden aus der Sicht eines Programmierers die einfachste Architektur eines speichergekoppelten Multiprocessorrechners. Die einzelnen Prozessoren eines SMPs sind über den Speicherbus (oder eine Kreuzschiene) mit einem gemeinsam nutzbaren Hauptspeicher verbunden. Dies erleichtert die Entwicklung von Anwendungen und Betriebssystemen verglichen mit NUMA Architekturen oder nachrichtengekoppelten Multiprozessoren.
Gleichwohl ist die Entwicklung eines SMP-Betriebssystems, selbst für eine so einfache Architektur wie den SMP noch eine große Herausforderung. Der zusätzliche Grad an Parallelität wirft dabei Probleme auf, die mit konventionellen Lösungen nicht mehr zu lösen sind, da das Aus- / Anschalten von Unterbrechungen (interrupts) nur prozessorlokal wirksam ist.

Der L4 Mikro-kern ist einer der schnellsten derzeit verfügbaren. Er steht auf folgenden Einprozessorrechnern: MIPS, Intel x86, Alpha und Arm zur Verfügung. Verschiedene Partnerprojekte implementieren Anwendungen, Betriebssystemkomponenten bzw. eigenständige Betriebssysteme auf L4-Basis wie zum Beispiel:

1. *L^{4}Linux*[1] ist ein Linux Betriebssystem, welches als Applikation im Nutzermodus auf dem L4-Mikrokern abläuft.

2. *DROPS* [1] ist eine Echtzeiterweiterung zu L^{4}Linux, welche den gemeinsamen Betrieb von gewöhnlichen, wie auch speziellen Applikationen, die Rechenzeitzusicherungen benötigen, ermöglicht. Für erstere wird L^{4}Linux, als Betriebssystemserver verwendet. Letztere können auf spezialisierte Server zurückgreifen.

3. *Saw Mill Linux* [2] ist ein Multiserver Linux Betriebssystem. Bestimmte Betriebssystemdienste sind dabei als einzelne, eigenständige Server realisiert, die nach Bedarf zusammengefügt bzw. ausgetauscht werden können. Diese erhöhte Flexibilität kann u.a. dazu verwendet werden, auf besondere Situationen zu reagieren. So kann das Betriebssystem etwa auf den Verlust der Netzspannung, z.B: falls der Laptop vom Strom genommen wird, reagieren und von hoch performanten auf stromsparende Server wechseln.

4. *Mungi* [3] ist ein Einadressraum Betriebssystem. Alle Programme teilen sich dabei einen einzigen grossen Addressraum. Dies hat zur Folge, da/3 Verweise auf Speicherbereiche globale Gültigkeit haben.

Diese Studienarbeit untersucht, welche Auswirkungen ein symmetrischer Multiprozessor auf das Design und die Implementierung eines L4 SMP Mikrokerns hat. Insbesondere wird der Einfluss des SMP auf die grundlegenden Abstraktionen des L4 Mikrokerns untersucht: Threads, Addressräume und Interprozess Kommunikation (IPC).

1 Introduction

In the past years, shared memory multiprocessors became widely used as midrange servers and even as high end workstations. Symmetric Multi Processors (SMPs) form the simplest architecture of shared memory computers from the programmers point of view. Several equally functioning processors are connected through the memory bus (or a crossbar) with memory modules. Accesses to those modules occurs in a unique manner for all processors. This makes the development of applications and operating systems much easier than in Non Uniform Memory Architecture (NUMA) processing or message coupled multiprocessors.
Never the less, the explicit parallelism of the applications and operating system servers, makes the multiplexing of the CPUs much more difficult because, first the number of CPUs are increased and second the invariant that when disabling the interrupts no other program beside the one disabling the interrupt will interfere is no longer true.
This problem leads to the challenge of how to design and implement efficient Operating Systems for SMPs in particular and Multiprocessors in general. However this thesis will deal with SMPs only.
The L4 μ-kernel has proven to be one of the fastest microkernels for several uniprocessor systems like MIPS, Intel x86, Alpha and Arm. Several partner projects are building applications and OS components or even whole operating systems on top of L4:

1. *L^{4}Linux*[1] is a user level implementation of the Linux operating system which runs as a normal application in user mode.

2. *DROPS* [1] is a realtime extension to L^{4}Linuxthat allows to run a mixture of normal Applications and specialized ones, requiring certain guarantees of computation times to do their work, on the same CPU. The first kind of applications use L^{4}Linux as the operating system server. The last kind can rely on specialized servers.

3. *Saw Mill Linux* [2] is a decomposed multi-server Linux operating system. Certain operating system services are implemented as several independent servers that can be composed and exchanged on demand. The increased flexibility of such a system can i.e. be used to adapt the system to special situations dynamically. When i.e. the Laptop is unplugged, the

[1]TU Dresden:
http://os.inf.tu-dresden.de

[2]IBM Watson, Universität Karlsruhe, TU Dresden:
http://www.reseach.ibm.com/sawmill

[3]University of New South Wales:
http://www.cse.unsw.edu.au/ disy/Mungi

high performant servers can be swapped out and exchanged by special power saving ones.

4. *Mungi*[3] is a single address space operating system. All applications share a single huge address space which has the immediate consequence, that pointers holding an address are valid through the entire lifetime of the system.

Both, the increasing relevance of multiprocessors and the applicability of the microkernel approach motivates for a L4-SMP μ-kernel. This thesis will describe the prototypical design and implementation of L4 SMP mechanisms. In particular it will try to answer the question which extensions have to be introduced to the basic uniprocessor concepts: threads, address spaces and inter process communication.
The following two sections give a brief introduction to the L4 uniprocessor μ-kernel and to such parts of the Intel Pentium hardware that are relevant for SMP μ-kernel design and implementation. Those two sections are included in the paper to give a brief introduction to the environment, the remaining part of this thesis is based on. Readers that are familiar with the Pentium hardware and the L4 μ-kernel should skip these sections and continue in section 4.

2 The L4 uniprocessor μ-kernel

Monolithic kernels suffer from their inherent complexity and inflexibility. Changing the behavior of a single component, like the paging policy for example, requires at least recompiling the entire system. Faults and errors in the new component might propagate to other components and thus may effect the whole system. Ensuring fault tolerance of the system, has to take into consideration the entire operating system and not only the changed parts. Even parts of the system that are not depending on the changed components might suffer. Frequent errors in the new component (like an uninitialized pointer) may result in a system crash.

Multiserver operating systems help to tackle this problem. Each component of the system is implemented as a server, running in user level like any application does. Almost every modern processor implements at least two modes: supervised and user level.

- **User mode:** Applications run in user level. They are protected from being modified by other applications and from modifying other ones. Only a subset of the instructions of the processor is available in user mode. None of these operations can be used to harm or to compromise other applications or to monopolize the operating system as a whole. I.e. disabling the interrupts (with `cli`) cannot be allowed to be used in user level, because if this instruction is used by an application, the timer interrupt is disabled as well and the operating system may not get control of the processor in time.

- **Supervisor mode:** Code running in supervisor (or kernel) mode has full control of the entire system. It can use all processor instructions, even those that might compromise the system (when not handled with care like if the system runs in an endless loop while interrupts are disabled with `cli`). In supervisor mode, the kernel code can switch between different applications.
  When an application generates an exception, i.e. dividing by zero, the processor switches from user to supervisor mode, invoking a kernel method to handle this error. The handler eventually aborts the application, whilst the remaining applications in the system stay alive. However, if the same exception would occur while running kernel code, the system will crash.
  On an Intel Pentium exceptions in supervisor mode will invoke a handler method as well, but if the error cannot be repaired (i.e. a division by zero), the system will crash due to an abort of the kernel.

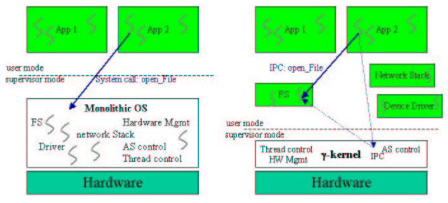

Figure 1: Monolithic Kernel vs. Microkernel

While monolithic operating systems run all their components in supervisor mode, multiserver OS components are executed in user mode. Just opposite to a monolithic kernel design we can place each component in a separated and protected address space [4]. Errors in one of the server components might crash this particular server but none of the others. Thus the rest of the system will survive. In particular a corrected version of the formerly crashed component can be installed on the fly and tested for correctness. Altogether, we have an environment, where we can test the components incrementally, making the system more robust.
The mechanism to exchange servers on the fly increases the flexibility of the system. Special servers can be loaded to adapt to new situations, like the power example described in the SawMill part of the introductory section 1, more than one server per OS function can coexist in the system at the same time, allowing the system to adapt to the requirements of specific types of application. The same system can be used to support a large database and a personal digital assistant PDA. In the first case high performing, large servers are combined, while in the second case lightweight, power and resource saving servers might be the better choice.
However two problems still remain: First, as described above, only a subset of the processor instructions are available in user mode and second, some part of the operating system has to remain in supervisor mode to multiplex the processor among the user level system servers and applications. Furthermore it has to handle exceptions and interrupts. This is exactly what a μ-kernel does.

The first μ-kernel approaches were constructed by extracting as much of the operating system's functionality out of the kernel and put this functionality into as many separate user level servers. The remaining part of the kernel should be as general and flexible as possible to allow a large variety of OS services to coexist. The separated components of the OS were protected against each other, in the same way as applications are isolated on top of a monolithic kernel. Provided the μ-kernel is correct. Faulty servers may crash but they can no longer affect other correct ones.

However, the first generation μ-kernels like Mach failed! The large communication overhead between the extracted components that was introduced by the μ-kernel mechanisms like IPC, made its use for a multiserver OS unaffordable.

[4]However, we can place more than one component into one address space if needed due to performance reasons.

Learning that lesson, the development of a second generation of μ-kernels had begun. While the first generation tried to extract several parts out of an existing monolithic kernel, L4 was designed from the scratch. The goal was to find a minimal but high performing set of mechanisms, implemented in the μ-kernel, with which any reasonable policy can be build on top. L4 implements only three basic abstractions to achieve this goal: Threads, Address Spaces and Inter-Process Communication (IPC). Threads and address spaces are used to multiplex the CPU and memory. Inter-Process Communication (IPC) is used to overcome these protection borders introduced by address spaces in a save manner. In the following subsections these three fundamental mechanisms and their usage for the uniprocessor L4 μ-kernel are described.

2.1 Threads

In L4, threads are small entities of code that can be executed concurrently. In contrast to the event model, threads are created and live until they are explicitly deleted. Synchronization between different threads has to be done explicitly through IPC for example. The opposing model, the event model can be shortly described as follows. On an event, a thread is created, computes and dies when it has finished its work. In between or when dying the thread might trigger further events that leads to the construction of further event handling threads. Multiple of those events can coexist in the system and are handled in parallel. In the event model, the threads do not explicitly communicate with each other, they use the events instead to trigger the service needed. Instead of synchronizing with other threads implicitly, in the event model, those events implicitly synchronize concurrent activities. I.e. in a matrix multiplication, multiple events are triggered by the compute thread to activate the working threads. When finished computing its part of the multiplication, the working thread generates an “I am finished” event and the compute thread collects all this events and if all working threads finished, it can combine the results to the global result of the matrix multiplication. So what was achieved in this example is a barrier synchronization through events. In opposite to a thread model, the worker threads are created on the first and die on the second event. However, since the threads in the event model does not store state information at the time they die on an event, the state has to be included into those events. While in the thread model it makes a difference whether an IPC (i.e. a call to a server to trigger some action) is send in the if or in the else path of a client’s code (i.e. if (condition) IPC: add (a, b) else IPC: subtract (a, b)). In the event model, dependent on the condition, one of two different events, representing the condition has to be triggered (i.e. condition true request add, condition false request sub events).
Because of this, L4 uses the thread model instead of the event model.
The information needed to manage and control a thread is stored in its Thread Control Block TCB. In the TCB, the current state of a thread is stored, i.e. if it is ready to run, waiting for another thread or currently communicating to another thread through IPC. Within a thread switch, the general purpose registers, the stack and instruction pointer are stored into the TCB of the current thread and reloaded from the TCB of the thread to run next.

2.1.1 Scheduling

To be flexible, L4 does not implement certain high level scheduling policies, but keeps this to the responsibility of a user level scheduling server. However due to performance reasons, fine grain scheduling is done by the kernel. The user level scheduling server can parametrise this fine grain scheduler for each thread under its control through the system call thread_schedule. Those control parameters for the μ-kernel’s dispatcher (fine grain scheduler) are located in the TCB as well.
A thread can voluntarily release the CPU by calling thread_switch. This call can be parameterized to switch to a specific thread or to let the dispatcher decide for the next thread to run.
In this case the dispatcher will chose the ready thread with the highest priority in the system. The priority of a thread is one of those above described control parameters that is stored in its TCB.
In addition, each thread has a period of time, the timeslice, that limits the time, the thread is allowed to run on the CPU. When the timeslice expires, the dispatcher is invoked and will chose the next ready thread of that priority in a round robin manner. So to sum up, ready threads with the same priority share the CPU in rates specified by their timeslices as long as no higher priority thread gets ready.

2.1.2 Thread Creation

Lthread_ex_regs is used to modify a thread’s instruction pointer, stack pointer and pager (see below). Threads are created by setting the instruction pointer to a valid value (Note, address spaces are only the first 3GB of the available hardware space. So above 3GB a user level instruction pointer is invalid because this area is reserved for the kernel). Lthread_ex_regs works only for threads, sharing the same address space. So the first thread has to be created in different way. This is done when the address space is created. In the next μ-kernel version, threads will be created through the explicit operation thread_control.

2.2 Address Spaces

“What is an address space? To answer this question we take a whole portion of nothing and call it an address space. In addition we let threads operate in it and if they touch somewhere in that nothing, it is up to the operating system to back the touched location with a page of physical memory.”

When in the Intel Pentium processor the protected mode and paging is enabled, the memory management change from direct handling of physical memory to virtual memory. The physical memory is split up into equally sized portions called frames. Each frame is 4KB large and aligned, i.e. starts with an address that is a multiple of 4KB.
The addressable area (4GB for a 32 bit address bus) is split up in 4KB regions as well. Those regions are called pages. We now introduce another level of memory: Virtual memory that is the addressable area, so consists of pages. Note the size of the pages are processor dependent, the Intel Pentium offers two different sizes for example: 4 KB and 4 MB, but for simplicity reasons, address spaces and mapping is explained for 4 KB pages only.
The page tables are used to store a map containing all information needed to translate an access to a virtual page, i.e a virtual address to a physical frame that backs that location. Accesses to a virtual page are automatically translated by the Memory Management Unit (MMU) into accesses to a physical frame by parsing the page tables. The Intel Pentium hardware for example has two level page table scheme. The virtual address (sometimes called linear address) is divided into three parts: an offset (the lower most 12 bits), and two indices (10 bits each) to address the first and second level page table entries. Starting from the Page Directory Base Address register (PDBA), that contains the offset of the first level page table, called page directory, the upper 10 bits of the address are used to find the page directory entry. This entry points to the starting address of the second level page table (simply called page table). The mid 10 bits of the virtual address index a page table entry that points to a 4 KB page frame. The lower most 12 bits of the page table entry are used

to decode several page attributes i.e. if the page is read only, or if the frame is present at all.
Note, not all pages have to be backed by a frame (entry is not present). Neither has to exist a translation to all the physical frames. Each page may map to (i.e. be backed by) either zero or exactly one frame. In the first case, we say the page is not present and if accessed, the hardware will raise a pagefault. Multiple pages may point to the same frame but one page may point to no more than one frame.
With this, the entire address range of the processor can be used, though not all addresses need to be backed by a frame at the same time. In addition each thread or group of threads may have its own virtual memory, called address space. When switching to a thread that does not share this address space, the page tables are exchanged and with this a new mapping of pages to frames is established.
In the paragraph above, mapping is used in its mathematical context. The term mapping was used to describe pairs: page to frame. Now we will use the term mapping for an operation on pages. We will see, that there is not much difference. Above, the term described a relation between virtual to physical addresses, i.e. pages to frames. Now we use the term for describing a relation between two pages, i.e. between two virtual addresses and construct an initial mapping concerning the physical addresses.

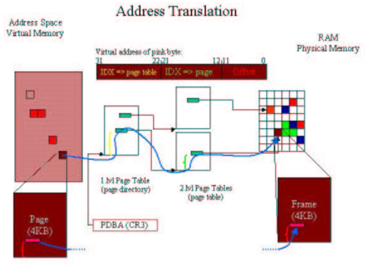

Figure 2: Virtual to physical address translation

L4 reaps benefit of the virtual memory concept to construct its address spaces. In a monolithic system it is up to the operating system to ensure, that the mappings of two address spaces do not overlap involuntarily. When each address translates its pages to a disjunct set of frames, there is no way for the first to corrupt the data in the pages of the second address space.
In L4, the construction of address spaces is up to user level paging servers, threads that run itself in an address space. Assume by magic, these address spaces get some data, i.e. a mapping (mapping in the formerly described sense) of some pages, backed by frames, was established. Address spaces can now be constructed by sharing pages. The threads that share the pages with the new address space are called this address space's pagers. Those pagers can only share pages owned by themselfs, i.e. only those backed by a frame can be shared. This operation is called mapping. It is performed by the μ-kernel on behalf of the pager. When a page is mapped to an address space, the μ-kernel parses the page tables, like the MMU does, to find the frame of the page to be mapped and creates a corresponding translation in the target's address space's page tables to exactly that frame. As a result, the page is now visible in both address spaces.
To avoid magic in the μ-kernel (because this is really hard to implement), we construct an initial address space called sigma 0 and establishes a mapping to all available frames (i.e. to those not used by the μ-kernel itself) by hand.
From this point on mapping works from address space to address space and operates on pages, i.e. virtual addresses. For the user level pager, frames are no longer visible.

To sum up, address spaces cover the entire address range of the processor, the entire virtual memory. However not all virtual memory locations (pages) may be covered by physical memory (frames). They exist per thread or per set of threads and multiple address spaces may coexist in the system. Two address spaces, with non overlapping mappings, that is taken an arbitrary page from one and another from the second address space, these pages will not be backed by the same frame. In that case a protection border is established, because threads in the first cannot corrupt the second address space's memory.

2.2.1 Mapping and Granting

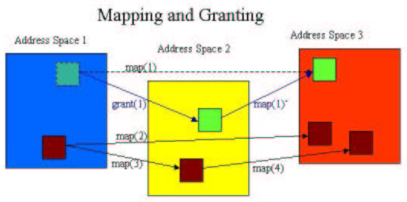

Figure 3: Mapping and granting pages

In L4 based systems, address spaces are constructed hierarchically by mapping (i.e. sharing) or granting (i.e. donating) pages from one address space to another. The entire hierarchy with sigma 0 as the root pager, is built up by mapping or granting pages from address space to address space, so from virtual to virtual address.
Despite those operations, map and grant and an additional operation to undo a mapping: unmap, the construction of this hierarchy is done by user level threads. Mapping works such, that a thread in one address space, the mapper, invokes the μ-kernel to map pages, that are in its address space to the destination's thread, the mappee's address space. To make this operation secure, both partners, the mapper and the mappee have to agree on the mapping, which makes this a synchronous operation. In addition, the mapper specifies the page that will be mapped and the mappee has to specify the destination where to map to. This has to be done to avoid interferences, i.e. a mapper maps a page to a location where it may harm the execution of threads of the mappee's address space, like when mapping a data page to a code page. Because of the similarity of the control requirements, mapping and granting are implemented as special message types of IPC (see below).

2.2.2 Unmapping and Pagefaults

By accepting a mapping, the mappee automatically agrees, that the mapper can revoke this mapping any time. A mapping can be revoked or weakened with the system call fpage_unmap. Weakening a mapping means that the page is not removed, but that the access rights are changed to read only. Other architectures allow

independent access rights for reading, writing and executing, so weakening in those architectures would allow write only, execute only, read execute, write execute and so on access right combinations.
In contrast to mapping or granting, unmapping has to be asynchronous. This means that the unmap operation will be performed immediately and will not wait for the mappee's agreement. This is necessary to avoid that the client (the mappee) postpones the unmapping request for pages formerly mapped by the server (the mapper) because of a denial of service attack by the client or simply by not listening to unmap requests. The asynchronous unmapping is for example being used by pagers implementing swapping. Before the memory pages are swapped out to disk the pages are unmapped, so that the clients of that pager can no longer use the page.
With these three operations: mapping, granting and unmapping, any policy can be implemented for constructing address spaces. But one point remains open: What happens if a thread accesses a not present page? As described above, a pagefault is raised by the hardware when either the page is not present or was mapped read only and is written to. The μ-kernel receives this page fault exception and translates it into an IPC to the pager of the faulting thread. This pager can be specified through the systemcall lthread_ex_regs and is a thread being established to handle pagefault messages. With this mechanism, any reasonable paging policy can be implemented.

2.2.3 Address space creation

Address spaces can be created and deleted through the system call task_new. This systemcall automatically initializes the data structures needed for an empty address space and creates the first thread in this address space, which then can create the other threads with lthread_ex_regs. L4 Version X was build around a task concept. A task is an address space with a set of threads, sharing this address space. The next μ-kernel Version 4 X2 will weaken this concept by creating the address spaces of the threads implicitly. Threads that are supposed to share the same address space can be created by specifying the thread, whose address space to share with.
The first thing this newly created thread will do when being started is to raise a pagefault on its code page, since the address space was created empty. This pagefault is send to its paging thread which is now able to fill the address space with content using the mapping IPC.
Up to now, we have learned about how to multiplex the CPU and memory and how address spaces establish protection between threads of different address spaces. Threads within the same task can interact easily via their shared memory. However how to interact with threads outside an address space?

2.3 Inter-Process Communication

Inter-Process Communication (IPC) is a mechanism to overcome the restrictions introduced by address spaces. An IPC enables that the two communicating threads can exchange data (a few bytes only, several strings, or map or grant memory pages). IPC is synchronous, which means both partners have to agree on the communication. In particular the first blocks until the second is willing to perform the communication. While waiting for the partner thread to start the send / receive phase of the IPC, this thread will be inactive. To avoid that one of the communicating thread can compromise the others data, the receiving partner has to specify the destination, where to map or copy the data, while the sender has to specify which data to transmit. In addition, both partners can specify two timeout values. The first one specifies the time that the sender / receiver is willing to wait for the partner to get ready for the transmission.
The second timeout limits the time for the partner's pager to handle page faults that may happen during an IPC. While the first timeout allows to react to a never responding threads, the second is needed to avoid attacks by a never responding pager.

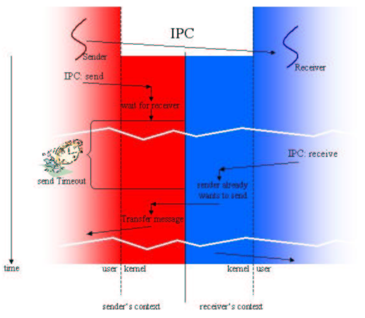

Figure 4: Uniprocessor IPC path

With this mechanism communication between untrusted partners (like a client with a server) can be established. A typical scenario is a call (i.e. an atomic send and receive operation) from the client to the server with timeout infinity, because the client trusts the server to do the job, followed by a reply and wait (a send to the specified thread followed by a receive from any thread) from the server with timeout 0. The server does not trust its clients and will discard the message if they are not listening to the answer (timeout 0). In this scenario, the send phase of the call from the client corresponds with the receive phase from a reply and wait operation from a previous job to trigger the job on the server side. When the work is done, the send phase from the server's reply and wait operation corresponds with the receive phase of the call. After answering, the server is waiting (ready to receive) the next job, while the client that was waiting for the servers answer is released and continues.
Though IPC is required for inter task communication only (intra task communication could also be handled though shared memory), IPC works for intra task communication as well. Since most client-server protocols require only very short messages, special optimizations are done for very short messages that fit into registers.

2.4 Mirroring and Control of the Processor

The preceding sections showed how the CPU and memory of a processor can be multiplexed. However the μ-kernel has to fulfill another purpose: mirroring the processor features.
Since these special features are highly platform dependent. Some are even processor stepping specific (like the performance monitoring facilities), I will describe only another two examples: Interrupt and Floating Point Unit multiplexing.

- *Floating Point Unit:* L4 multiplexes the Floating Point Units FPUs to enable concurrent accesses of several threads. This requires to save the state of the floating point unit including all registers on each thread switch. The next time, this thread gets the CPU, the registers have to be restored again.
  The Intel x86 hardware allows the FPU to be secured from accesses through others. Any access to a secured FPU raises

a fault. L4 catches this fault and if the same thread has caused the fault, no registers have to be saved. Only when a different thread accesses the FPU, the registers are saved to the old thread's TCB and restored from the new one's.

- *Interrupts:* In a monolithic operating system, the bottom half interrupt handler is directly attached to an interrupt. This handler is usually invoked in a disabled interrupt state to avoid another interruption before the handler can set-up the information for the upper half handler.
  In a μ-kernel based system, multiple of those handlers may coexist. Some may require direct control of the interrupt line, like the bottom half handlers or specialized interrupt handling routines. Some others may be satisfied with an acknowledgment that a specific interrupt has occurred, like the top half handlers.
  L4 installs an internal interrupt handler for each line, that translates the interrupt into an IPC from a "hardware thread" to the handling routine, that waits for that interrupt (receive from hardware thread). With this mechanism, it is achieved that high priority threads are preferred compared to lower priority interrupt handlers. We may even distribute interrupts to a pool of handler threads. This is similar to a multithreaded server.

Up to now, we know how the uniprocessor L4 μ-kernel works and that it is easy to build operating system servers on top of L4 like user level pagers and drivers. We know threads and address spaces as two major μ-kernel objects and IPC as a communication mechanism between threads. The next section should give an understanding of the SMP hardware features of the Intel Pentium processors which is the targeted platform for the implementation part. For the design section, the Pentium hardware is general enough to get a basic idea has to design a μ-kernel for other architectures, too. Though certain parts may be different the behavior of the Intel Pentium hardware can be generalized to other SMP platforms, i.e. similar orders of magnitude in the latencies of memory accesses can be considered on cache coherent SMPs, though the exact implementation of the coherency protocol and of course the exact access times may differ a little bit.

3 Pentium Hardware Support for Cross-processor Communication

Intel x86 SMPs couple the 2 to 8 functionally identical CPUs through a memory bus with main memory modules. Bus accesses are guaranteed to be atomic. In addition, the bus can be locked to establish atomic read-modify-write operations. Memory accesses are not ordered with respect to the other processors, but the accesses of any single processor are observed by all the others (i.e. they are snooped by the cache coherency protocol).

3.1 The Memory Hierarchy

Two fast memory modules (Level 1 and Level 2) represent processor local caches. Whenever an access misses in the L2 cache, the request is transmitted through the bus and carried out in one of the memory modules. In addition, a bus snooping MESI protocol establishes cache coherency. Both caches have 32 byte wide cache-lines which are usually organized in 8 banks of 4 bytes each.

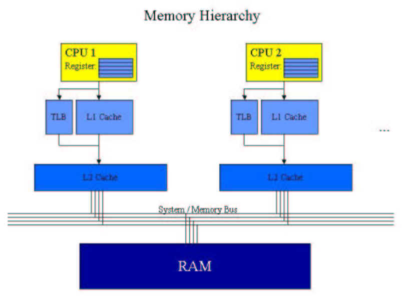

Figure 5: Intel Pentium SMP memory hierarchy

The L1 caches are virtually indexed and physically tagged and in size and associativity exactly that large, that the index information can be extracted out of the physical bits [5] of the virtual address. On a read or write, the index of the address is put in parallel into the L1 Cache and the Translation Lookaside Buffer TLB, that caches the least recently used translations from virtual to physical addresses. When hitting in the TLB, the physical address is compared to the tags of the selected set of the cache and the data is delivered on a match. In the case that none of the L1 tags matches, i.e. the L1 cache misses, the address is put into the physically indexed and physically tagged level 2 cache. If the TLB misses, that is when no translation exists for this virtual address, the MMU is triggered to parse the 2 level deep page tables [6] and refills the line in the TLB. The page table entries itself are cached in the L1 and L2 cache and read from it on a TLB miss. This is possible because the L1 cache is at the same time virtually and physically indexed (see above).

3.1.1 TLB Shootdowns

The processor local Translation Lookaside Buffers are used to cache the least recently used translations of virtual to physical addresses.
When switching to another address space, the same virtual page may point to a new physical page frame. Without TLBs, switching the page directory is sufficient, because an access to the virtual page requires to walk the page tables and translates to the correct, that is the new physical frame. Taking TLBs into account, a translation of the same virtual page to the old physical frame may be cached. Accessing the page would hit in the cache and finds the translation to the old frame, instead of walking through the page tables. This inconsistency may result in compromising the old address space. To avoid this, the cached TLB translations of the old address space have to be removed, i.e. the TLB has to be flushed.
To avoid this flushing, tagged TLBs were developed. In addition to the two addresses a number representing the address space ID is stored in the tagged TLB. Complemented by that ID, the virtual addresses differ and thus need not to be deleted explicitly. Untagged TLBs like offered by the Pentium processors do not store this additional information. Due to this lack, virtual to physical translations may be cached, though the corresponding entries in the page tables have been changed. To keep the TLB consistent, the modified translations have to be flushed so they are reloaded the next time, the page is accessed. Especially when switching to another address space, the TLB have to be flushed and reloaded.

[5] the lower 12 bits of the virtual address specify the offset within a 4KB pages.

[6] Intel calls these page tables page directory for the first and page table for the second level

The Intel Pentium offers two instructions, to invalidate parts of the TLB: `invlpg` invalidates a single line, `mov cr3 reg` loads a pointer to a page directory (specified in reg) into the page directory base register (PDBA or CR3) implicitly flushing the entire TLB. Both instructions work only processor local.
As an immediate result, deleting one or more page table entries (Mapping a page to an occupied location is specified to unmap the old page first. The implementation does no real unmap, but overwrites the old page table entry. Switching to a new address space can be viewed as deleting the entire page table.) has to result in triggering the invalidation of the corresponding translations in all TLBs. Even those of the other processors. This operation is called TLB shootdown.

3.1.2 Cache Coherency

The Pentium processor has two caches L1 and L2 to hide memory latency. The caches are kept coherent even in SMPs. This means, that reading a value from the caches will always return the last modification, even when done by another processor. A bus snooping MESI protocol establishes this coherency.
MESI stands for the possible states of a cacheline: exclusive modified (M), exclusive unmodified (E), shared unmodified (S) and invalid (I).
Initially the cache is in invalid state (I). When reading, the line is filled and becomes exclusive unmodified (E). Writing to an exclusive or invalid line would change the state to exclusive modified. In the latter case the contents of the line is loaded from the underlying cache / memory before the write is performed. When a second processor P2 reads a line that is in exclusive unmodified (E) state in P1's caches, i.e. was read before, the read is snooped by both processors. Instead of P2 changing to exclusive unmodified, both processors will change the state to shared unmodified (S) noting this line is in some other processor's cache, too. When writing to a shared line or a line not in the local caches (exclusive unmodified in P1's cache when P2 is writing), any processor, that holds this line (P1), snoops the write and invalidates it before the write is performed. When reading or writing to a line that is modified in the other processor's caches, the write is snooped similar to the last example. But before invalidating, the processor holding the line (P1) writes it through to memory, so P2 can read or write to it. Modern chipsets no longer write through to memory and then read again from the memory chips, but in parallel snoop the contents written back to memory from the bus into the cache. For the read operation, both cachelines are in shared state afterwards.

3.2 MESI, Cache and TLB Performance

In the last sections we learned how caches work, how the MESI protocol establishes cache coherency and what to gain from TLBs. This section should give an overall idea of how that hardware performs. I will not describe any pathological cases since those special cases occur very infrequently if at all in reality.
The measurements were done on a dual PIII SMP with a 450MHz CPU, a 16 KB 4 way set associative level 1 and a 512 KB level 2 cache. The processors were plugged in a 440BX board with 100 MHz memory bus frequency and a 64 MB RAM.

3.2.1 Cache and Memory access times

To get a feeling about the order of magnitude of accessing cached data compared to non cached date, I consecutively read a dword from memory addresses, that correspond to the first dword of different cachelines and measured the delay of that operation with a CPU internal cycle counter. To avoid side effects, I read one byte of each page in advance, assuring that a valid translation is cached in the TLB. To generate L1, respectively L2 misses, the caches were flushed by reading some other data that clashes with one to be used for the measurement. To get the overhead of the measurements itself, I also read a dword directly from a register. After preparing the caches with loading the data, followed by to flushing of specific parts I measured:

1. reading 256 dwords from register
2. reading 256 dwords that hit in the L1 cache. Each dword was read from a memory location that would be stored in a separate cacheline.
3. reading 256 dwords / lines that misses in the L1 cache, but hits in the L2 cache.
4. reading 256 dwords / lines that misses in the L1 as well as in the L2 cache.

on a single processor while the other one was idling without accessing the memory bus in between. Table 1 shows the cycles spent for the entire loop, a single iteration of the loop, the time needed for reading the line, without the overhead of the measuring loop which are the cycles per iteration subtracted by the iteration cycles for register reads. The last column is added to get an idea what is happening on the memory bus. The cycles spend for reading a line are transformed according to the frequency of the memory bus. 1 memory bus cycle equals 4,5 = 1 * 450 MHz / 100 MHz processor cycles. So while only one tick is available to transmit data for the memory bus, 4.5 ticks have to be spent in the CPU.

	cycles per 256 lines	cycles per iteration	cycles per line	**memory bus** cycles per line
Register	571	2,20	0,00	-
L1 hit	815	3,18	0,98	-
L1 miss	2491	9,73	7,53	-
L2 miss	11462	44,77	42,57	9,46

Table 1: Average times for cache and memory accesses of a dual PIII 450 MHz

The results show, that accessing data within the L1 cache can be done in one cycle. Actually two cycles are needed for L1 accesses, but two requests can be handled in parallel which results in a 1 cycle average. Missing in the L1 but hitting in the L2 cache takes about 8 to 10 cycles, delaying a program by one order of magnitude, if the reads are not hidden behind the computations. Data missing in the L2 cache, has to go through the memory bus to the RAM modules, leading to a delay of almost another order of magnitude (another factor of 5).
This result shows, that it really pays to reuse data in the L1 cache. In the PII and PIII processors, certain prefetch operations are included into the processor core that may be used to load data into the L1 or L2 cache in advance, provided the bus is free and the prefetch command can be setup early enough.

3.2.2 MESI performance

Table 2 shows the performance of the MESI protocol for the same experiments like above with the difference that the same data is read or written to by two processors. In order to do the measurements, the cache on the first processor was prepared to hold 2 times 256 dwords exclusive unmodified in the L1 cache, i.e. a read on this processor would hit the line, and 2 times 256 dwords (lines) in modified state. Like in the experiments above, the addresses of the data is chosen such, that each of the dwords accessed would be

cached in a separate cacheline. Given that preparation the measurement code will find the data in the caches in the right states.
One after the other, the following operations are performed by the second processor:

1. 256 dwords are read, that were prepared to be in the first processor's caches in exclusive unmodified state. So the same 256 dwords were read before by processor one. The lines in both caches change to shared state. **E to S**.

2. As above, except that the 256 dwords, that formerly were in exclusive unmodified state in the first processor, are written by the second processor. This results in invalidating the lines in the first and setting them to modified state in the second processor's caches. **E to M**.

3. The next two experiments operated on modified data, i.e. on data that was written to by the first processor. First, the dwords are read by the second processor, leading to change the state of the lines in both processor's caches to shared unmodified. **M to S**.

4. Second, the dwords were written by the second processor, leading to an invalidation of the lines in the first processor and the caching in modified state on the second. **M to M**.

5. The last experiment operated on shared lines, i.e. the dwords are present because of former reads in both processor's caches. In this experiment, the second processor writes the dwords leading to an invalidation of the lines in the first cache, and changing the lines to modified state. In contrast to the M to M case, the lines have not to be read into the caches before. **S to M**.

	cycles per 256 lines	cycles per iteration	cycles per line	**memory bus** cycles per line
E to S	16902	66,02	63,82	14,18
E to M	13212	51,43	49,23	10,94
M to S	23482	91,72	89,52	19,90
M to M	23448	91,59	89,39	19,90
S to M	13211	51,43	49,23	10,94

Table 2: Measurement of the MESI protocol timings for the same loop

The results show, that snooping a read on a unmodified line requires about 1.5 the time of a L2 miss, because the cache has to read the line from memory and to wait for the second cache that is snooping the read, to modify the state of its lines from exclusive to shared. When a modified line is involved, the costs double. In that case, the line has to be written back to memory when snooping the write, even if the second cache would modify the line again. Note that the dirty lines have to be written to memory, before they can be read into the second processor's cache in the 440 chipset, the written data cannot be snooped from the bus in parallel. Writing a line results in reading it first, because the modifications may be done to parts of the line only.
In the last experiment, writes to a shared line were measured. In this case, the contents of the line has not to be written back to memory, but the cache lines have to be invalidated in the first processor's caches.

3.3 APIC

The Advanced Programmable Interrupt Controller (APIC) extends the hardware interrupt mechanism by additional interrupt lines, a clock and the possibility to trigger software interrupts in other CPUs. To accomplish that, the APIC network consists of three different units: the local APIC, the IO-APIC and a three line APIC-Bus. Each CPU has a build in local APIC that allows to handle processor local devices and contains a cycle counter derived from the memory bus clock that can be programmed to trigger timer interrupts. Each IO-APIC chip receives additional 240 hardware interrupts (int 0 .. 15 are reserved) and delivers them through the APIC-BUS network to the least loaded or one selected CPU's local APIC. Least loaded in this case means that the processor with the lowest priority is selected. When receiving an interrupt, this priority is raised. Compared to the 21 cycle short message for delivering interrupts to a fixed location (see below), a 34 cycle bus message is used that is send to all processors. Each processor's local APIC publishes its inverted priority for this case and that with the lowest priority wins and receives the interrupt. In the same way that the IO-APIC triggers an interrupt in a CPU, the local APIC chips deliver Inter Processor Interrupts (IPIs) to other processors [2].

For our purpose, IPIs are the most interesting feature of the APIC, because they can be used to trigger some action in the other processors. The sender, processor A initiates the inter processor interrupt by writing its number into the Interrupt Command Register (ICR). The destination processors can be selected through a mask in the upper most byte of the ICR. Setting the n^{t}h bit selects the n^{t}h processor as the target. In this mode, any group of the at most 8 processors can be selected as a target for the IPI. There are further modes for machines with more than 8 processors, but in those, the any-cast facility is lost, so either the APIC can deliver the interrupt to one specific or all processors. The local APIC hardware acquires the APIC bus, selects the target and initiates the interrupt. On the target processor B, the currently running thread is interrupted (provided the interrupt was not masked) and the interrupt handler code is activated. Though IPIs are software initiated, for the target processor they are identical to hardware interrupts because they happen asynchronously to the executing code. To acknowledge the interrupt, the handler code on B can send an End of Interrupt (EOI) message back to A, however A has to poll for this answer if it is required. If A does not need an acknowledgment it may continue.

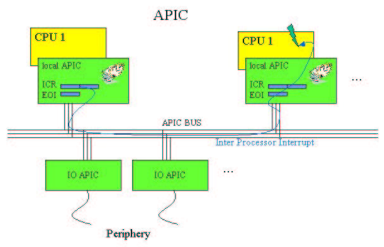

Figure 6: The Intel Pentium APIC and its network

3.3.1 IPI Performance

To evaluate the IPIs, two numbers are of interest: obviously 1. the time required to deliver the IPI and 2. the time, the target processor is interrupted for handling the IPI interrupt.

1. **IPI delivery time:** The APIC bus is three lines wide and runs at a frequency of 16 MHz. One of the lines is used as a clock, the remaining two for data. A short interrupt message, i.e. a message to a fixed destination requires 21 cycles on the APIC Bus: 4 for Arbitration, 3 to specify the mode, 4 (8 bits) for the interrupt number which can be up to 256, 4 (8 bits) to specify the destination plus some checksum, status idle and normalize cycles. On a 450MHz PIII this would sum up to roughly 590 processor cycles (Note one APIC bus cycle corresponds to 450 / 16 $\approx$ 28 processor cycles).
   I measured the time needed until the interrupt is delivered to the target. To achieve this, the destination processor B polls for the interrupt request register (IRR) in which a bit per interrupt is set, if the request comes in. The target processor B copies this information into some memory location, where the sender polls for. While polling and copying, the interrupts are disabled, so that the polling is not interrupted. A measures the time between writing the ICR, i.e. sending the IPI and receiving the acknowledge in the memory location updated with the IRR contents.
   But why not measure in the interrupt handler? At first it is not for sure that the cycle counter is synchronized. So reading at the same time from the two processor local counters (on A and B) may return different values. On the other hand, the destination processor B has to finish the currently executed instructions, flush the pipelines, refill the pipelines with the interrupt handler code that has to be looked up first in the interrupt descriptor table, change the privilege level and so on. All this activity take quite some time (roughly YYY cycles) and varies because of many reasons. Compared to that, triggering the send in the ICR takes about 40 cycles plus another 20 for polling for the IRR to change. Additionally we have to modify a shared line on B and reread it on A, but for this we already know the time needed, which sums up to 50 + 90 = 140 cycles. This sums up to additional 200 cycles.
   The measurements resulted in 1230 - 1250 cycles to deliver an IPI and detect this in the above described manner. The estimated value was 590 + 200 = 790 cycles. The remaining 450 cycles (note this are 16 APIC Bus cycles only) can be explained by the work, done in the APIC state machine, before the IPI is send over the APIC bus.

2. **Interruption time:** the second value of interest is the time, the destination processor B needs to handle the interrupt triggered by the sender A. To measure that, a thread reads the clock in a closed loop. When being interrupted, the interrupt handler stores the value read by the user level thread before being interrupted and sets the loop condition to finish after the clock was read one more time. The difference of the two values minus the time needed to store the first and set the break condition is the time spent to enter and exit supervised mode and invoke the interrupt handler. If in addition, the handler code itself reads the cycle counter, a distinction between the two phases: enter kernel mode and activate handler, and returning to the interrupted thread again, can be distinguished.
   In the Pentium processor, the hardware pushes the eflags register, stack and instruction pointer of the user level thread on the kernel stack. The EFLAGS register contains the state of the processor immediately before the interruption. For the loop break condition, the carry flag is used. The handler can modify this flag by resetting it on the kernel stack, where it was pushed when entering supervisor mode. When returning to the user level thread, the EFLAGS register is popped automatically. In contrast to the EFLAGS register, the general purpose registers are not pushed on the stack automatically. So they can be modified by the handler code directly when being used. The current system timestamp is read into the processor registers immediately before the interrupt and immediately after entering the handler code giving the cycles needed to switch from user level code to kernel mode into the handler code.
   For the kernel entry phase I measured between 120 to 160 cycles. The exiting phase costs between 170 to 250 cycles. Including the privilege level change from user to kernel and back to user mode.

In total roughly 1030 (1230 - 200 for the polling) cycles after the sending processor A writes its ICR to trigger the IPI, the interrupt arrives on the destination processor B. B then enters supervisor mode (provided the interrupt was not masked) and activates the interrupt handler 120 to 160 cycles later. So about 1200 cycles after processor A triggered the IPI, the handler code gets activated. The currently running thread on processor B is interrupted for 290 to 410 cycles plus the time needed for executing the handler code. No further interrupts of a lower or equal priority can be received by B's local APIC until B ends A's interruption by writing an arbitrary value to the End Of Interrupt EOI register of the local APIC. So to sum up, every 1200 cycles (2 μs) an IPI can be received by the target processor B, that interrupts the currently running thread for about 400 cycles (1 μs). All at all, up to one third of the available processor cycles may be spent in handling inter processor interrupts. Two things can be concluded from this:

1. Inter processor interrupts should not happen more frequently than once per 100 μs to get no more overhead than 1% of the processor cycles.

2. Kernel entries and exits should be avoided if possible.

4 Outlook of the Paper

The last two sections gave a brief introduction to the L4 uniprocessor μ-kernel, on which the following design is based on, and to the Pentium hardware that is relevant mainly for the implementation part of this thesis (see Section 8).
The remaining part of this paper is structured as follows:
Section 5 describes the major design goals of the L4 SMP μ-kernel and section 6 the design itself. Section 7 summarizes the changes to the system calls that are effected by the design decisions. In section 8, the implementation issues on a Intel Pentium hardware are described. Sections U and V explain the open points and conclude the paper.

5 Goals

There are six major goals for the SMP μ-kernel design:

1. *Flexibility*

2. *Generality*

3. *Performance*

4. *Compatibility*

5. *Transparency*

6. *Non Transparency*

The *Compatibility* and *Transparency* goals are required to provide backward compatibility to existing uniprocessor code. Such software should not need to be modified to run properly on the SMP μ-kernel. The *compatibility* goal ensures the interface compatibility, while the *Transparency* goal ensures the compatibility of the SMP μ-kernel mechanisms compared to the uniprocessor ones. In particular this means that a task designed for an uniprocessor μ-kernel should not have to care about SMP extensions.
The *Non Transparency* goal, however, enforces the support of SMP specific applications (like parallel numeric programs - for example matrix multiplication) and servers (like network or load balancing servers).

5.1 Flexibility / Generality

The basic idea of the μ-kernel approach was to provide basic mechanisms to implement any reasonable policy in user level servers. To achieve this, the μ-kernel mechanisms have to be as flexible and as general as possible.
This particular goal has already lead to the design of the uniprocessor L4 μ-kernel and hopefully will hold as well for SMP μ-kernels.

5.2 Performance

An application or server, designed to run on a uniprocessor, should have the same execution time running on the uniprocessor machine as well as on a specific processor of a SMP.
The bottleneck of tightly coupled SMP architectures is the memory bus. It connects the processors and DMA chips with the memory modules, but only one processor or DMA chip may acquire the bus at one time. Because of memory bus contention, tasks accessing memory on an SMP may lead to even worse results than if running on two separate uniprocessor machines. Some preliminary ideas dealing with these effects are presented in [1, 3]. The solution of this problem however is out of the scope of this paper.
Never the less, the μ-kernel mechanisms should not suffer from SMP side effects and therefore have to be carefully designed and implemented.

5.3 Compatibility

Existing L4 application and OS servers should run on the SMP μ-kernel without modification. This means that the system calls have to keep their functionality and binary interface or at least we have to provide backward (uniprocessor) compatible interfaces.
Some applications may rely on uniprocessor specific synchronization mechanisms, like disabling interrupts. Those unconsciously designed applications will not be supported any longer in the SMP μ-kernel and may fail.

5.4 Transparency

In the large, it should be possible to construct and implement applications and system servers such, that how its threads are assigned to the processors of an SMP does not effect their functionality. In the small, a thread should see no functional difference whether its partner thread is on the same or a different processor.

5.5 Non Transparency

Though at the first look, this goal contradicts to the *transparency* goal, both have to be taken into consideration when designing a SMP μ-kernel. While the *transparency* goal requires to hide SMP specific situations from existing uniprocessor applications to avoid modifications in the code, the *non transparency* goal requires to give user-level programs access to SMP specific features (like the migration of threads) in a secure and comfortable way.

6 SMP specific μ-kernel Design

The measurements of the cache MESI protocol show that the latency of accesses to shared data structures differ only slightly, assuming the data reads have to fetch from main memory. For cached data instead, the delay for cross-processor reads and writes cannot be neglected. This holds in particular for performant critical kernel mechanisms like IPC. Additional synchronization overhead adds to this, when synchronization is required.
In the Pentium processors, only read and write operations are guaranteed to be atomic. Some read-modify-write operations can be made atomic by locking the memory bus during their execution. However with none of those operations, more than one dword (4 bytes) can be modified [7]. So software synchronization primitives like spin locks and semaphores are needed when having to synchronize complex operations.

In the L4 μ-kernel only few “global” i.e. shared μ-kernel data structures, can be identified. Basically only mapping data including the page tables and the coarse-grain wakeup queues are of such type and have to be protected against cross-processor access. None of those structures are used by time critical operations.

Thread Control Blocks (TCBs) and the ready queues, require similar cross-processor access synchronization only if we decide to have fine-grain inter-processor scheduling (dispatching) of threads, that is if the dispatcher selects the next thread from a global pool of ready threads. In that case, we would have to explicitly synchronize every dispatching access to a TCB, even to the current TCB to avoid inconsistencies. If not, per-processor dispatching data structures and some IPC tricks (see below) would enable us to manage TCBs such that cross-processor accesses can be restricted to very special cases: cross-processor IPC, explicit migration to a new processor and cross-processor TCB accesses by the system calls lthread_ex_regs and thread_schedule. None of those operations – remember only cross-processor operations are listed – seems to be first-class time critical.

6.1 Processor Scheduling Granularity and Migration

When pinning a thread – from its birth to death – to one specific CPU, fine grain scheduling automatically works locally only. However, restricting a μ-kernel to this simple design contradicts the *flexibility* goal. In multiprocessor operating systems, several load balancing policies have been proposed, making use of thread migration in case of overload situations. To support those kind of policies, the μ-kernel has to offer some basic mechanism to migrate threads. At first the granularity of such a migration mechanisms is examined. This inflame when and how frequently threads may migrate. (The discussion of how to migrate threads is postponed for the time being.)

6.1.1 Implicit transfer through the dispatcher

Some older SMPs with small cache memories permitted quite inexpensive transfers, nowaday SMPs do not. Transferring a thread with a working set of 50 dirty lines and 50 read only lines – we

[7]Note: the Pentium string operations are implemented as a loop of a single operation. Each iteration can be set to be atomic, but not the entire loop

assume that these lines are at least in the L2 cache – would require roughly 90 cycles per dirty and 65 cycles per read only (exclusive) line which sums up to 7750 cycles (see section 3.2). Between 4500 and 7750 cycles (10 - 20μs) on a PIII dual 450 MHz SMP are required to transfer only the dirty (minimum time) or all (maximum time) lines of this thread's working set. It can be assumed that the threads to be migrated have most part of their current working set at least in the L2 cache. If not, it means that either this thread is not scheduled for a long time, or that its working set was replaced due to clashes with other frequently used threads. For the first case, this thread will not be the primary target to reduce the load of one processor. The latter case should be avoided by system design or by cache coloring techniques.
The cycles needed to transfer the working set can even be worse on machines with 4 or more processors because of memory bus contention, i.e. due to concurrent bus accesses of the remaining three processors (the source processor is not actively involved in this transfer and may run another threads in between). Caches contravene this delay, because accesses hitting in one of the caches and not triggering cache coherency actions, do not require to access the memory bus. In addition to that, there exists data that cannot be cached effectively (i.e. video stream data or accesses to a large database in a randomly or hashed manner). This analysis is e.g. corroborated by the analysis and experiences of the K42 project [5].
To sum up, executing a thread transfer more often than ones per 200μs seems to be unreasonable. This leads to

- ***Design decision 1:*** *Fine grain scheduling will not implicitly transfer threads to other processors. Migration has to be initiated by a user level scheduler.*

There are three immediate conclusions from design decision 1:

1. Dispatching and fine grain scheduling works processor local, this implies that ready lists and "short wakeup queues" have to exist per processor.
2. The system call thread_switch switches to processor local threads only. Specifying a non local thread works as if nil-thread was specified. In that case, the next ready and local thread is selected to run.
3. The μ-kernel has to offer some mechanism to let a user level scheduler migrate threads.

6.1.2 How to migrate threads

Concluding from *design decision 1*, the migration requests will be initiated by a user level server. On the other hand it is obvious, that some μ-kernel intervention is required, because threads are μ-kernel objects. We still have to decide what type of intervention is required for migrating threads.
The symmetry of the SMP architecture makes the decision on which processor to run a thread independent from special features of some processors in the system. Processor local vector computing units for example need not to be taken into account.
The load of the system is controlled by user level scheduling servers, that have to keep track of all relevant information i.e. indicators of the load of the processors by themselves.
Because migration decisions are triggered by user level schedulers, an in kernel migration mechanism has to consider the thread transfer, only.

When migrating a thread from one CPU to another, its TCB has to be deleted from all local data structures on the source processor and reinserted into the destination processor ones. Note, *design decision 1* lead to processor local ready and short wakeup lists for example.
One possible way is not to support a special migration mechanism, that moves existing threads around, but instead to use the thread creation and deletion mechanism. Migration would then mean to delete a thread on one CPU and create a new thread on the destination processor. When implementing migration in that way, the current state of the thread would be lost, so it has to be saved before the deletion and restored afterwards. However the kernel internal state is not available in user level, i.e. the list of sending threads, that are waiting for the thread to enter the receive path, is present inside the kernel only. A second counterargument is the time needed to create and delete a thread. Creating a thread requires to build up and initialize a new TCB, which costs some time. Additionally, the old TCB information cannot be reused without restricting the delete operation.
Third, the current kernel implementation does not support real thread deletion. A thread can be aborted by the μ-kernel or through lthread_ex_regs, but its TCB will never be released.
This leads to

- ***Design decision 2:*** *Migration has to be initiated by a system call.*

A system call is needed to migrate threads. Obviously this call has to be somehow protected to avoid attacks against the load balancing and scheduling system (i.e. by migrating all threads to one CPU). We extend the system call thread_schedule, that already controls the other scheduling parameters like timeslice and priority by a parameter to specify the current processor. Changing this parameter will migrate aimed thread to the specified processor.
The thread_schedule can only be used by threads with a higher MCP. The maximum controlled priority (MCP) value of the scheduler thread specifies the highest thread priority it is allowed to control. Only the scheduling parameters of threads with a lower priority than the MCP value can be modified with thread_schedule. Normal, non scheduler threads, will have a MCP of 0, so they are not allowed to modify the scheduling parameters of any thread in the system. In the next μ-kernel version, a specific scheduling server is specified for each thread. Only this thread is allowed to modify the scheduling parameters and by this to migrate.
So far, a thread can be migrated by a system call, initiated from a user level scheduler. It will not be migrated by the dispatcher of the kernel. Next we have to discuss some situations where the thread itself may want to migrate, i.e. to be mobile.

6.2 Thread Mobility at μ-kernel Level

Mobility is a thread's ability to migrate itself between several processors.
Design decision 1 already rejects that μ-kernel dispatching migrates threads and also concludes in a way to let user level scheduling and load balancing servers migrate threads by extending thread_schedule.
However, two possible scenarios remain: restricted explicit thread transfer and implicit mobility on IPC.

6.2.1 Restricted Explicit Thread Transfer

Explicit thread mobility would enable application threads to migrate to other processors by themselves. Though nice to have, i.e. to automatically parallelize a problem for computation and join again to compute the results, this functionality has to be restricted to protect certain processors (i.e. those dedicated to realtime threads).
However since migration costs are about 1 to 2 orders of magnitude higher compared to a call to a local scheduling server and since more frequent migrations than once per 200μs result in drastically

down performing the system, the additional short IPC per migration to contact a user level scheduler is neglectable. From the policy point of view it is always preferable to contact a user level server when this additional performance loss does not hurt. This server can implement an arbitrary migration policy and i.e. protect processors dedicated to realtime threads from being burdened with additional load.
To sum up, explicit mobility will not be implemented by the μ-kernel. Instead a user level scheduling server should be contacted which is able to migrate the thread using the thread_schedule systemcall.

6.2.2 Implicit Mobility on IPC

A somehow similar scenario is implicit mobility due to an IPC. For performance reasons, a server or a client thread may want to automatically migrate to the partner's CPU to be able to process on local and presumably cached data. Another reason might be to improve the performance of following IPCs to the partner. If both partners have to communicate frequently in the near future, the higher startup costs for cross-processor IPCs can be saved, when migrating to the partners processor at the first IPC. Especially for short IPCs, where the startup costs dominate the overall performance of the IPC, avoiding cross-processor IPC pays. For very long messages, which means that probably not the entire message fits into the caches and that some pagefaults may happen in between the transfer, the higher startup costs for cross-processor IPC are neglectable.

Implicit mobility would allow a thread to specify whether to automatically migrate to the partner's processor on IPC. A server thread that migrates on a receive would always migrate to its clients processors before executing its task. A migrate on send server would migrate to the client thread's processor after it processes its request on the last clients processor. The next obvious idea is to let the server decide on a case by case basis whether to migrate or not.
This example shows that mobility on send / receive is insufficient and that any other, more complicated mechanism would include policy. This makes the mechanism either to restrictive or not general enough and therefore contradicts to the *flexibility and generality goal*.

Provided the kernel implements an efficient detection mechanism for cross-processor IPC, an arbitrary mobility on IPC policy can be implemented by calling the scheduling server after cross-processor IPC was detected. This mechanism should even be preferred from the policy point of view, because the additional knowledge of the desired communication can be taken into consideration for the schedule. I.e. the scheduler may avoid to distribute communicating partners to different CPUs for load balancing reasons. This leads to:

- ***Design decision 3:*** *The μ-kernel will not implement implicit mobility on IPC, but support a fast cross-processor IPC detection mechanism.*

6.2.3 Detecting Cross-Processor IPC

Detecting cross-processor IPC is like a "two sided blade". On the one hand it is required to implement arbitrary mobility policies and fast adaptation to the new environment for mobile threads, on the other hand it opens a covert channel by allowing threads to mors a message only by the information on which processor they reside.
Section 9.1 describes the problem of covert channels a little more detailed. But since we currently do not know how to deal with covert channels while having an acceptable performance independent of a fixed security policy, we will ignore the problem for the time being.
When talking about cross-processor IPC detection, two questions have to be answered: When to inform the IPC partners, and what information to deliver?

When to inform the IPC partners? The IPC operation is implemented as a systemcall which means that usually supervisor mode is entered and the kernel IPC code performs the operation on behalf of the threads. The next kernel Version will implement an IPC path for some special flavors of the IPC in user level, but fall back to the within kernel IPC mechanism whenever needed. This means that independent of when to inform the partners, the current μ-kernel has to enter and exit supervisor mode. In Version 4, some cross processor IPCs may be detected in user mode, saving the time needed for the switch.
In general, cross-processor IPC can inform the communicating partners before the message is copied or afterwards. In between the copy sequence does not make sense, because no additional benefit can be gained compared to informing before the message transfer. A partly copied message is of no use and has to be copied again.
So two potential ways remain when to detect cross-processor IPC:
a) IPC signals the invoker that it has been a cross-processor IPC after the transfer is completed, and
b) IPC fails in the cross-processor case before it happens.
Conceptionally a) is a trap, while b) is a fault.

XP-IPC detection after the message transfer Mechanism a) extends IPC such, that it reports cross-processor IPC after it happened. Despite the additional costs for moving the message data to the destination processor's cache, the major overhead of cross-processor IPC is in the startup phase. For long messages, this startup costs becomes less relevant, so detection after the message transfer is not critical in those cases, assuming the message has to be delivered anyway. For short messages, the additional overhead of cross-processor versus processor local IPC has to be taken into account. In those cases, detecting cross-processor IPC after the transfer might be to late.

XP-IPC detection before the message transfer Compared to a), mechanism b) informs the invoker before the IPC happens. In the startup phase, the kernel checks whether IPC is going to cross the processor boundaries and fault in this case. If b) is implemented, any thread, even those that do not care about cross-processor IPC, has to implement a recovery mechanism for the cross-processor case. But this contradicts to the *compatibility goal* and requires code modifications to existing uniprocessor applications, which is unacceptable. To avoid this, the invoker has to specify whether cross-processor IPC should fault or not, so when enabled, the IPC will fail in the cross-processor IPC case. When disabled, this mechanism degenerates to a). Obviously, mechanism b) works only for the send phase of the IPC, that is on a send or a call. For a receive only IPC this mechanism degenerates to a).
To avoid the additional overhead of mechanism a), we will implement mechanism b) which can be degenerated to a), by disabling the faulting feature explicitly.

What kind of information will the detection mechanism supply? Basically there are three different levels in the amount of information that is useful to retrieve through such a mechanism:

1. the information whether this IPC was cross-processor or not.
   A single bit is retrieved that indicates if the IPC was cross processor.

2. the partner thread's processor number.
   I.e. the number of the processors that are participated in the IPC.

3. the partner's processor number plus some guaranteed time in which the partner is pinned to this processor. Which means that the processor number of the partner will not be changed within the specified time.

Retrieving thread pinning information The time information retrievable through the third option can only be retrieved through the partner's user level scheduling server. Of course the μ-kernel can be extended by a call to specify the time a thread is pinned. But in addition, the μ-kernel has to ensure that this time is not decreased and that no migration will happen in between. However this restricts thread migration drastically. Second, to retrieve the information from the user level scheduler, the kernel has to send the request and wait for an answer of this scheduling server which creates a not acceptable dependency between those two. Either way this option is unacceptable for a μ-kernel mechanism. In general, no information that can't be kept in the μ-kernel itself can be delivered.
Remaining, a decision between 1 and 2 has to be made.

Thread mobility For thread mobility there is no difference between 1) and 2), because the migration requests can specify the target processor implicitly through a target thread. For this purpose a protocol like "Transfer me to B's processor" should always be preferred to a processor based like "Migrate me to processor n", because then the additional knowledge of the thread B makes it easy for the scheduling server not to transfer B in between.

Adapting to thread migration Another scenario is when adapting to a thread migration. That is when a thread B is sometimes contacted from thread A. In this scenario two possible situations show up: B is migrated and has to detect on which processor it currently is i.e. to contact a local thread of a multithreaded server, or A has to detect the new position of B.
In the current μ-kernel version (Version X), B has to ask its scheduler for its current processor. In the next version, B can read the processor number of all threads of its task from the UTCB field: processor number[8]. So when detecting a cross-processor IPC to a formerly local thread, reading the UTCB field reveals the new processor.
For A the situation is somehow different, because B might not be in A's task so A cannot read B's UTCB field and B's scheduler might not be known to A. In this case, option 2 is required to find out about B's new location.

Drawbacks What drawbacks does the second option bear? First, we decided to detect cross-processor IPC before it happens. This means the message is aborted and that there are enough registers available to return the processor number. In the non faulting case this might be a problem on processors with very few registers, since the registers are in use for the message transfer. Then a single bit for detecting XP-IPC is preferable (a status flag i.e. the zero or equal flag can be used for this purpose). Despite this, transmitting only one bit compared to at least 4 bits for the processor number, reduces the bandwidth of potential covert channels, but only by a factor of 8. This leads to:

- ***Design decision 4:*** *The μ-kernel will offer a mechanism to detect cross-processor IPC in that way, that the sender of an IPC can specify to abort the IPC in the cross-processor case. In this case the kernel will return the processor number of the target thread.*

- ***Design decision 4a:*** *Covert channels will be ignored for the time being. The detection mechanism will be revisited for the covert channel problem.*

6.3 Address Spaces

Since L4 multiplexes the CPUs at thread granularity and not at task granularity, two threads of the same task can be running on two different processors in parallel. But how does the address spaces look like on this two processors? The *compatibility* and *transparency* goal require the same address space layout, however the *non transparency* goal leads to ask for the support of processor local data structures.
The *compatibility* towards the uniprocessor μ-kernel leads to another problem when looking at the above situation: How to deal with concurrent memory accesses?

6.3.1 Processor Local Data Pages

L4 allows to compose address spaces by mapping or granting pages into it. Beside the mechanisms for mapping, granting and unmapping pages, the decision which pages to map where and when is up to user level paging servers.
Shared memory multiprocessors can access any physical memory location from any CPU. Nevertheless the access time to different portions differ from fast for local memory to slow for memory that is far away and has to be accessed via some interconnect. This holds for NUMA architecture machines, but not for SMPs. Memory access in tightly coupled SMPs is uniform. However, a similar effect can be observed when caching is considered (though in another order of magnitude). Data in the local cache can be accessed fast, while data in another processor's cache may have to be written back to memory first leading to even worse performance than if directly read from memory (see section 3.2).
When local data can be accessed faster in general, there is the question for the μ-kernel to support local data, i.e. data structures that are visible on one processor only. The μ-kernel has no knowledge about application internal data structures, however the kernel knows pages and the application data is stored in memory pages. The mapping mechanism can easily be extended to support local data structures, i.e. local data pages. Each mapping is extended by a mask specifying on which processor the pages are mapped. In the flexpage structure, that specifies the pages to be mapped or granted, there is no free field to store this information. So either the *compatibility goal* has to be ignored in this case or an additional flexpage structure has to be introduced for the SMP kernel.
However, implementing this would require a page directory per processor per task and for the local data pages additional page tables per processor are required. With the same overhead in space, an address space per processor can be created using the existing mapping mechanism to build almost identical tasks, that share all, but some local data pages.
To sum it up, processor local data pages are not supported. If needed, this can be implemented in user level easily.

[8]The User Thread Control Block UTCB is a user level accessible part of a threads TCB required to implement fast user level IPC

6.3.2 Concurrent Memory Accesses

In general, concurrent memory accesses occur in the same way as on uniprocessor machines. The only difference is, that at the same time, more than one thread can have access to the shared data. Despite the atomic operations for reading and writing to memory, there is no hardware mechanism to synchronize accesses, so this has to be done in software. In the uniprocessor L4 μ-kernel a thread can be interrupted at an arbitrary position in the granularity of single instructions [9], i.e. when a higher priority thread gets ready to run. Because of this, even in the uniprocessor system, synchronization for concurrent data accesses is required.
Several monolithic operating systems offer a large variety of synchronization primitives like semaphores, locks and monitors to synchronize concurrent accesses to shared objects like files. The large variety of different synchronization primitives and policies that exist, requires a user level implementation of such synchronization primitives i.e. in a library or in special synchronization servers, but for *flexibility* reasons not in the μ-kernel.
However in a uniprocessor system, atomicy and synchronization can be achieved by disabling the interrupts for the time of the operations to be atomized. In a uniprocessor system, the invariant holds, that at most one thread is actively running on the CPU. If this thread has a higher priority compared to all the others between which to synchronize, only this one thread of those would be scheduled. Some applications construct synchronization primitives relying on this fact. However, since disabling interrupts has to be secured to avoid that a thread can monopolize the system and since scheduling should be orthogonal to synchronization and its policy might even change when changing the user level scheduling server or the kernel primitives, those implementations are simply incorrect and short sighted and will not be supported by the μ-kernel. Software relying on that type of synchronization may fail on a SMP kernel.

- ***Design decision 5:*** *The μ-kernel will offer no special support for processor local data nor for concurrent memory accesses. Both can be implemented in user mode, easily.*

6.4 Cross-Processor IPC

The *compatibility goal* and the *transparency goal* almost automatically enforce the next design decision:

- ***Design decision 6:*** *IPC works cross-processor as well as processor local. Both flavors have identical functionality. IPC can be used transparently. That means the communicating partners don't have to care about on which respective processors they run. However they can care about that using the cross-processor IPC detection mechanism explained above.*

This of course means that the API and ABI has to be identical for the non detection case.

7 Affected Systemcalls

7.1 L4 Version X0

- **ipc:** Due to the lack of control bits in the input parameters, the IPC abort functionality will be postponed until Version 4 X2. However on return the zero flag indicates if it was a cross-processor IPC.

- **thread_switch:** Thread_switch can be used to switch to the specified thread, donating the remaining part of the own timeslice. Since fine grain scheduling is processor local, this call will not switch to a non local thread. In such a case, the kernel dispatcher will look for the next local and ready thread.

- **thread_schedule:** The input parameters are complemented by the processor number [10] Bits 16 - 19 of the param-word hold the processor number. When specifying a valid processor number 1..k (where k is the highest available processor number in the system), the thread will be transferred to that processor. Specifying the processor number zero means no transfer happens.

- **l_thread_ex_regs / task_new:** Both calls can be used to create threads, the last one to create the first thread of a new task. Threads will be created locally on the current processor of the creator and have to be migrated to another processor on behalf through thread_schedule.

- **kernel info page:** In the kernel info page, the μ-kernel publishes the highest available processor number (k processors are numbered 1 .. k, 0 means invalid) in the 8-bit word at relative address 0xB8. The higher 24 bits are reserved for future use, i.e. in n-way SMPs and NUMAs.

7.2 L4 Version 4 X2

- **Schedule:** specifying a valid processor number (1..k where k is the highest available processor number) transfers the destination thread to that processor. Specifying the processor number zero means that no transfer will happen.

- **Thread_switch:** Thread_switch can be used to switch to the specified thread, donating the remaining part of the own timeslice. Since fine grain scheduling is processor local, this call will not switch to a non local thread. In such a case, the kernel dispatcher will look for the next local and ready thread.

- **IPC:** when the cross-processor fault bit f, bit 12 of the send word 0 (snd.w0), is set, the IPC is aborted in the cross-processor case. In this case the error code 7 is raised with offset 0. In this case the processor number of the destination thread is returned in XXX. After completion, bit 11 of the receive word 0 (rcv.w0) is set if the IPC was cross-processor.

- **LIPC:** since cross-processor IPC requires kernel intervention, LIPC will enter the kernel and setup the corresponding IPC operation in the cross-processor case.

- **Thread_control:** This call is used to create threads [11]. Threads will be created locally i.e. on the creator's current processor. They have to be explicitly transferred through thread_schedule on behalf.

- **UTCB:** The threads current processor number is published in the UTCB at relative address 12 (24 for 64 bit processors). This field is read only, which means a write to this field leads to undefined behavior and perhaps even to the destruction of the task. The processor number is a value between 1 .. k where k is the highest available processor number.

[9] Some of the Pentium loop instructions like the string operations are interruptible after each iteration

[10] The Bits 16 - 19 are defined to be zero in the Version X uniprocessor API. Starting with processor number one and defining zero to no migration will happen provides compatibility with the uniprocessor L4 μ-kernel.

[11] Note lthread_ex_regs which is now exchange_registers does no longer create threads

- **kernel info page:** In the kernel info page, the μ-kernel publishes the highest available processor number (k processors are numbered 1 .. k, 0 means invalid) in the 8bit word at relative address 0x20.

8 Implementation

Up to know we discussed the design of a SMP μ-kernel, summed up by the design decisions in section 6 and described the effects and modifications to the system calls, i.e. the API of the μ-kernel in section 7.
In the following chapters, we take a look at the implementation of a L4 SMP μ-kernel on an Intel Pentium SMP. In detail we discuss the implementation of the cross-processor IPC and its implications on the thread control block structure. We consider what has to be done to migrate threads and how to implement local data structures efficiently. In the next subsection we discuss how to bootstrap a SMP μ-kernel.

In a uniprocessor system, there is a helpful invariant, which was used to implement most system call functionality in the L4 uniprocsessor μ-kernel: *When running inside the μ-kernel, i.e. system call code, there is no actively running thread on the CPU. Disabling the interrupts ensures, the code is not interruptible.*
Obviously this is not true for multiprocessor system, however a similar invariant can be constructed: *When running inside the μ-kernel, there is no actively running thread on the* current *CPU. Disabling the interrupts ensure, the code is not interruptible by* processor local *threads.*
The main difference is, that in the latter case some other non local thread can be actively running and in a SMP system modify the system call's data concurrently (provided the thread is allowed to, i.e. is executing some kernel or supervised code). The μ-kernel itself has to deal with this concurrency.

8.1 Single versus per processor μ-kernel instances

When looking at a SMP as a uniprocessor with some few additional processors plugged in, implementation decisions were derived from uniprocessor kernels. The SMP features are viewed as extensions to a uniprocessor design. On the other hand, when looking at a SMP from the multiprocessor perspective, the SMP is a very tightly couple of processor nodes. Having this in mind, the implementation decisions were derived from multiprocessor kernels. The following section discusses both for deciding how to bootstrap a μ-kernel.
When bootstrapping a μ-kernel on a multiprocessor machine, we have to decide whether to start an instance of the μ-kernel on each processor or node, or one single instance for the entire system. Like mentioned above, the first idea is derived from multiprocessors, while the second one is derived from an uniprocessor μ-kernel design.

The single nodes of which a message coupled multiprocessor is constructed are quite similar to uniprocessors. A CPU is connected to memory that is locally accessible only. The several nodes are connected with some kind of interconnect, sending data packages around. In the cluster case, the individual nodes are real uniprocessor machines like PCs. In such an environment, the first idea would be to use uniprocessor μ-kernels on each single node and add some features (if possible as a user level server to keep the μ-kernel flexible and small) to handle cross processor activity.
The SMP however has a very tightly binding of the CPUs in the system to the memory modules. Single nodes can't be identified. In such a type of system on the first look, a single instance of the μ-kernel would be preferred instead. In the following two subsections, the two approaches are compared for building an SMP μ-kernel.

8.1.1 Per processor μ-kernel instance

A per processor instance of the μ-kernel requires to separate the machine into p mostly independent parts, where p is the number of processors in the system. Each single instance takes control of its single part of the system, i.e. its physical memory and its CPU. *Design decision X* already implies to have only processor local fine grain scheduling and a certain processor locality for threads. So in a per processor instance SMP μ-kernel, threads are only local kernel objects. Only in cases of migration, a thread has to be transferred to another processor, i.e. to another μ-kernel instance. Since threads are local kernel objects, an interface has to be implemented in the μ-kernel to send and receive threads.
Address spaces on the other hand are system global kernel objects. L4 allows to schedule two threads that share the same address space on two processors at the same time. But with this, inconsistencies in the address spaces may occur, i.e. through concurrent mapping and unmapping operations. Note not the operations itself bear the problems, because threads can trigger concurrent operations on address spaces in a uniprocessor system, too, i.e. when sending two long IPCs containing overlapping mappings. Since long IPCs are interruptible it is not guaranteed which of the two will win and get its page mapped.
The problem of the concurrent operations is located in its implementation. Instead of an atomic modification of the page table entries when mapping a page, like in the uniprocessor, the mapping operations (that modify the page table entries) may be carried out in parallel.
A per processor μ-kernel instance means that the maintenance data for address spaces including the page tables are kept locally. This requires some mechanism keep the maintenance data consistent, like a MESI style protocol that was already presented for caches coherency.
On the other hand, any one single instance of the per processor instance μ-kernel can be exchanged easily, provided the interfaces remain unchanged. This can be done as easily as migrating all the threads to the other processor's kernels and back again after the exchange happened. This works because each μ-kernel is completely independent from the others and therefore increases not only the availability of the system but as well the fault tolerance. A single faulting instance will no longer affect the entire system.
Especially in SMPs, global while consistent data comes for free. Even worse, the partitioning of the physical memory in the SMP is not architectonical but has to be done artificially. A faulting μ-kernel instance, i.e. one that references an invalid pointer, may crash the entire system, because no architectonical protection between the memory modules are available. These arguments speak for a single instance μ-kernel for SMP systems.

8.1.2 Single μ-kernel instance

In contrast to a per processor instance, a single instance of the μ-kernel can be chosen to control the entire system. In this model, the μ-kernel implements per processor data structures as needed, i.e. for threads and scheduling, but also shares global data. Shared resources are managed globally, so no additional kernel interfaces nor any partitioning of the system is needed.
Migration (see below) can be implemented easily as a cross processor operation on internal objects, accessing them through the shared memory. However, exchanging the μ-kernel on the fly is a little more complicated compared to having per processor instances. In the worst case this requires to shut down the system and reboot

with the new kernel, saving all threads in advance. Additional care has to be taken when implementing the μ-kernel because of the single point of failure, having a single instance only.

When considering caches, a similar situation like in a NUMA (Non Uniorm Memory Accesses) case shows up. However the access times are at least one to two orders of magnitude lower. Section 3.2 showed that accessing data in the local caches is about a factor of 50 faster compared to accesses to main memory. This has to be taken into account when implementing a highly performing SMP μ-kernel.
For a per processor instance μ-kernel, this is hard to achieve, because the interfaces have to be specified in advance and for a wide range of different architectures and processors. Having a single instance however, those tricks can be hidden in the implementation.
For a SMP μ-kernel we can conclude, that the benefits gained from a single instance implementation dominate those of the per processor instance implementation. In this case, it is preferable to construct the SMP μ-kernel starting with a uniprocessor implementation. So to sum up:

- ***Implementation decision 1:*** *The μ-kernel will be implemented as a single instance.*

8.2 Bootstrapping and Startup

In the following section we look at how to bootstrap the μ-kernel on an SMP system. Implementation decision 1 concluded to have a single instance only, i.e. it is up to this single instance to initialize and bring up all the processors in the system to a state, where the threads can run on. The Intel Pentium hardware selects one single processor to be activated when starting the machine. This processor is meant to bootstrap the system and to initialize the other processors. After this happens we end in a state, where a single user defined thread (besides sigma 0) is started while all the other processors are waiting to get a thread migrated to them. This user defined thread and its address space (called booter task) is meant to bootstrap the remaining basic user level servers needed. Note, the bootstrap processor plays a special role only during the bootstrapping process. After that it acts like any other processor in the SMP.

8.2.1 An example bootup sequence

The following bootup sequence resulted from extending the L4 uniprocessor assembler kernel (codename lemon pip) with the required SMP functionality.
When the processor starts up, one processor, the bootstrap processor, wins and loads the basic kernel components. It switches to protected mode, initializes the processor local tables needed for paging and initializes the basic interrupt handling routines. At this point, the kernel sends a startup inter-processor interrupt (IPI) to all the other processors in the system (called Application Processors AP) to boot them up. While depending on the processor and the booting environment (DOS, Grub) the L4 μ-kernel may be started on the bootstrap processor in real or in protected mode, the application processors start in real mode. The first part of the started code switches to protected mode, using the tables initialized by the bootstrapping processor. After synchronizing with all the other processors, the bootstrapper initializes the memory pool and from this point on all processors start their initialization of processor local data, requesting memory from the system global pool. Since the Interrupt Descriptor Table IDT can be shared, it is sufficient for the bootstrap processor to fill in the descriptors needed to find the handlers and the system call routines, provided the handler and system call routines are reentrant.
After each processor has initialized the dispatcher thread (note, the dispatcher thread needs to be processor local due to *design decision 1*) and all local kernel tables are initialized, the application processors idle, waiting for a thread to be migrated from the bootstrap processor.
After synchronizing with the application processors to know that the system is up and waiting for threads to be migrated, the bootstrap processor starts the booter task which is meant to bring up the remaining system servers. The specified synchronizations are of a barrier type, i.e. all processors wait until all processors in the system reach this barrier. After this synchronization point, only the bootstrap processor is running and the other processors wait until being released explicitly. When no synchronization can be achieved, that is when not all processors reply, the μ-kernel can take some recovery actions i.e. to restart a single not responding processor or to mask this processor out by reordering the processor numbers such, that the highest available processor number can be decreased.
During the bootstrap process, the application processors spin for their processor numbers using an atomic exchange and add instruction. With this number, the local APIC chip is programmed such, that the address of the local APIC, i.e. the destination to which the IPI is sent, matches the processor number. This trick allows to find a processor easily through its processor number and to detect failures in the bootstrapping process.
The current implementation detects missing processors and panics, but future implementations might take steps to solve the problem, i.e. starting a special and correspondingly privileged correction code.

8.3 Cross-processor IPC

Cross-processor IPC is a special case of IPC, in which the two communicating partners currently reside on different processors. Design decision 6 already concluded to have transparent cross processor IPC which means that there should be no difference in using cross processor IPC compared to intra processor IPC. Cross processor IPC was intended not to migrate the communication partners. Design decision 4 described a detection mechanism that can be set to abort the IPC if it would be a cross processor IPC and if doing so return the partner's processor number. If not aborted, the μ- kernel signals whether the IPC has been cross processor or not.
The remaining section is structured as follows. First we discuss the API and different message types of the uniprocessor IPC. Then it is told how IPC was implemented in the uniprocessor L4 μ-kernel and it is to identify what no longer works when switching to the cross processor case. Finally we start a general discussion of how to perform an operation on a destination processor, i.e. how to trigger cross-processor operations. Given this, we can construct an IPC path for cross processor IPC with the leading goal of keeping the normal case: short intra processor IPC, high performant. This section is concluded by the evaluation of the described IPC path and what it needs to abort such a cross processor IPC (i.e. when lthread_ex_regs is called to one of the communicating partners).

8.3.1 Uniprocessor API and message types

Though IPC is implemented as a single system call, it can be used in different flavors. In general we can differentiate between *sending* and *receiving* a message. In addition two atomic "send and receive" operations are supported:
1. *call* and
2. *reply and wait*.
A *call* contains a send phase immediately followed by a corresponding receive. A *call* is used whenever we want to contact a server and wait for its answer when its job has been done. This operation has to be atomic to allow the server to reply with a zero

timeout. Meaning the replying *send* (that corresponds to the *receive* of the *call*) fails and the server will probably discard the message if the receiver is not listening to it.
If a *call* would not be an atomic operation, the client might be interrupted before it is able to setup the receive and the server might reply before the client was able to listen.
The second operation *reply and wait* is a *send*, followed by a *receive from anyone* (called open receive in contrast to close when specifying a target to receive from). This operation is atomic, too, but only for performance reasons, i.e. to safe the additional kernel entry / exit between the *send* and *receive* operation.
Though we have those atomic operations, we can differentiate between the send and the receive phase of the IPC. In more detail we have a phase where one of the two partners is waiting for the other to get ready to send or receive and a phase where the message is transmitted. So to speak we have a wait for receive or a wait for send and a transmit phase in the IPC.

Timeouts The time, a thread is willing to spend in each of these three phases, can be limited by two timeouts per thread. The sender specifies the send timeout, that is the time he is willing to wait for the receiver to get ready and (since Version 4) a transmit timeout to limit the transmission phase. In the same way, the receiver parameterizes the IPC with a receive timeout, i.e. the time he is willing to wait for the sender, and as well a transmit timeout. The effective transmit timeout is the minimum of both, the sender's and receiver's transmit timeout. In L4 version X, the transmit timeout degenerates to a pagefault timeout, i.e. the time a thread is willing to wait for the partner's pager to handle pagefaults that occurred within IPC.

Message Types IPC can be used to send untyped words, as well as typed elements like strings or memory pages that will be mapped or granted. As long as no more than 3 dwords (in Version 4: 4 dwords) are transmitted, i.e. the message fits into the registers, the message is transferred directly. Messages larger than this have to be stored in memory and therefore addressed indirectly. When transmitting those messages in the IPC, called long IPC, page faults in the sender's as well as in the receiver's address space may happen. Long IPC messages are interruptible and the copy routine runs in the uniprocessor μ-kernel with interrupts enabled. Because of this, the page fault IPC can be setup as usual, interrupting the current message transfer process until the pagefault is handled.
A detailed specification of the message format and the IPC parameters can be found in the L4 Reference Manual for version X, respectively Version 4 [**?**, **?**].
Above, we discussed how to use IPC. Now we will look at how it is implemented in the uniprocessor L4 μ-kernel (codename Hazelnut).

8.3.2 Uniprocessor IPC Path

As described above, IPC can be divided into a send operation and a receive operation to setup the IPC, followed by the transmission phase. We assume, we entered the IPC system call code in supervisor mode i.e. through an `int 30`, or `sysenter` instruction and the interrupts are disabled. Starting in this state, we now discuss the send and receive path separately. Note the only difference for the atomic send and receive operations was that we do not leave supervisor mode between the end of the send path and the beginning of the following receive path.

Send Path At first we have to check whether the target is present and valid. Therefore we need to find the TCB of the target thread which was specified through its thread id *dest*. To check whether the TCB is valid and present [12], we check the *myself field* of the target TCB. This field contains the thread id of the owner of the TCB, in this case the target thread's id. Thread ids in L4 are basically composed in two fields, a number defining the thread itself and a version number to allow to make the ID unique in time. Threads with the same thread number share the same TCB, which means that only one thread can exist per thread number, so no two threads with different thread ids but equal thread numbers (i.e. where only the version field differs) may coexist in the system. In Version X this thread number is composed of a number defining the task, plus a thread number within that task. A thread id of 0 means this thread is not present. To sum up, we check the *myself field* of the target thread against 0 and compare it with *dest* and abort IPC when either the first comparison resulted in a match, that is the *myself field* was 0, or no match is achieved in the second comparison.
Next we set the sender's (A's) partner field to the receiver's (B's) TCB, indicating that A wants to send to B.
If at this moment B already is waiting for us, we can directly transfer the message. To find out whether this is the case, we check for B's state, if he is waiting and for A, i.e. if B is in a closed receive from A, or in a open receive. If so we can prepare A to send, if not, we have to wait for B to get ready to receive.
If A has to wait for B, A sets its send timeout by inserting itself into the wakeup queue of the system. Second, A enqueues itself to B's send queue, a linked list of all threads that want to send to B. This is done to avoid actively checking whether B got ready to receive from A. Actively checking, i.e. polling would be a correct implementation, however it generates unnecessary overhead in the system. Instead of letting A actively poll for B, A is inserted into B's send queue indicating the wish to send. B, when getting ready to receive, dequeues A from this send queue and reactivates it. Never the less we have to denote that A is prepared to send but waiting for B. This is done by setting A to polling state, not indicating that A will poll actively, but that A is waiting for B and like if actively polling will be activated again when B gets ready to receive. See the receive path below to see how B activates A.
Now A switches to B with a normal thread switch. When A gets activated again (Note polling was implemented as an inactive operation) two things may have happened: either B got ready to receive and did wake up A, or the timeout occurred. In the second case, the IPC operation is aborted with completion code timeout, in the first case A has to dequeue itself from the wakeup list and prepare for transmission. If B was ready to send to A before, we would end up in exactly this state, so now we continue with preparing A to send.
At this point, we lock A and the receiver B to denote that both are currently within an IPC operation and that no other thread will send them in between. Since on a uniprocessor system it is fully symmetric whether the sender or the receiver transmits the message (see the transmit path below), we chose the sender as being actively copying. Therefore A has to be in a ready to run state, like if no IPC is happening. To differentiate this, the lock prefix is added, so A is set to locked running, meaning it is ready to run, so it can be scheduled at any time, but no IPC can be send to it because it is locked.
Now we can transmit the message. The transfer finishes, either successfully or it was aborted due to an error or timeout.
If the message was transferred successfully and no receive phase follows, we set A to ready, enqueue it into the ready list and return with completion code success. Otherwise, we continue with the

[12] When no TCB is present, it is either zero filled or if no page is present and a pagefault occurs, a zero filled page will be mapped.

receive path (see below).

Here in short what was described above:

```
// send path
  if isSend() then
          from_TCB = getTCB(A)
          to_TCB = getTCB(B)

          if not exists(to_TCB) or to_TCB.myself $\neq$ B then
                  abortIPC ("target does not exist")
          endif

          from_TCB.partner = B

          if B is not waiting or not for me or open then
                  enqueueWakeup (A with A's send timeout)

                  to_TCB.enqueueSendQ (A)
                  from_TCB.state = polling

                  switchTo(B)

                  if returned with timeout then
                          abortIPC ("send timeout occurred")
                  endif

                  dequeueWakeup (A)
          endif

          // At this point, B is ready to receive from A

          from_TCB.state = locked running
          to_TCB.state = locked waiting
          enqueueReadyList (A)

          // A is now schedulable

          transmitMessage ()

          to_TCB.partner = A

          if error in the transmit phase then
                  abortIPC ("transmission error")
          endif

          if not isReceive() then
                  from_TCB.state = ready
                  enqueueReadyList (A)

                  returnIPC
                    ("message successfully transmitted")
          endif
  endif
```

Receive Path Again the source thread A sends to the current thread B, the receiver. Note that this is the corresponding receive path for the above described send IPC. When read as the receive operation that is part of a call, i.e. A formerly send to B, the letters have to be swapped: A receives from B.
The receiving thread B can specify its IPC to be open, i.e. he is willing to receive from any thread, or B may want to receive from one specific thread (A) only. We call the case, the receiver is waiting for a specific sender thread closely.
Remember, the sender A is in polling state if it is ready to send and waiting for B to enter receive state. Semantically it would be correct to prepare this receive state and wait for the next polling cycle of A to startup the message transfer. However this would result in frequent overhead of polling threads in the system. To avoid this, L4 implements no real polling, but checks and reactivates the inactively polling sender if the receiver becomes ready to receive.
To reactivate the sender we need to find it. When in closed IPC the sender is implicitly known. In an openly receiving IPC the sender has to be extracted from the send queue, a linked list of threads that are ready to send to B. If any thread is present in the send queue, we take the first one and specify it as IPC partner. If not we have to wait.
Therefore we check A, the sender, against nil, no thread was in the send queue, if it is polling, i.e. ready to send to us and finally if it is sending to us, i.e. if A's partner is B. If A is a valid ready to send to us thread, i.e. all but the first question was answered positively, we can reactivate A. If not we have to wait until A gets in that state and reactivate it then.
Like in the send path, we wait by enqueueing ourselves (B) into the wakeup queue, but now we have to take the receive timeout of B. Additionally B has to be set to waiting state before it can release the CPU through switching to preferably A if A is known at this time.
In this case, B is either woken up because its timeout occurred or by A because it is now ready to send to him. If the first case, the timeout, happened, we abort the IPC like in the send path with a timeout error message. Otherwise, we dequeue B from the wakeup queue.
If we found a ready to send thread in our send queue or directly specified, we have to reactivate it. We set its (A's) state to locked running, enqueue it into the ready queue, so it would be scheduled, dequeue A from our send queue and from the wakeup list that it entered while waiting for us to get ready. Next we have to prepare B for the receive by setting its state to locked waiting, i.e. B is waiting for receiving a message, but no other thread can send to him because it is locked. After that we switch to A.
When we return, because when A finished transferring the message it switches to B, we have to set B's state to ready, enqueue it into the ready queue and return from the IPC call.
Again here a shortcut of the above described:

```
// receive path
  if isReceive() then

        to_TCB = getTCB(B)

        if closed receive then

              from_TCB = getTCB(A)

              if from_TCB is an invalid TCB then

                    abortIPC ("Invalid source")

              endif

          else
              // open
              from_TCB = B.sendQueue.dequeueHead()
        endif

        if from_TCB not present or A is not polling or
           A is ready to send but not to B then

              enqueueWakeup
                (B with B's receive timeout)

              to_TCB.state = waiting
              to_TCB.partner = A (nil for open receive)

              switchTo (A)

              if timeout then

                    abortIPC ("timeout occurred")

              endif

              dequeueWakeup (B)

          else

              // A was ready to send to B
              from_TCB.state = locked running
              enqueueReady (A)
              dequeueWakeup (A)

              B.dequeueSend (A)

              to_TCB.state = locked waiting

              switchTo (A)
        endif

        to_TCB.state = ready
        enqueueReady (B)

        returnIPC ("message successfully transmitted")
  endif
```

The above described startup path of an IPC is an implementation of the version X API. Version 4 mainly differs in the message structure. We have to deal with slightly different thread ids for local threads, i.e. IPC between threads of the same task offer special means of optimization, but this is out of the scope of this paper because this special IPC, referred to as LIPC will not work cross processor. Additionally we have to include a transfer timeout. This will be done in the transfer path.

Transmit Path In the code described above, the sender is the active partner and copies the message to the target's address space. For uniprocessor μ-kernels this is a symmetric task, i.e. it does not matter whether the sender or the receiver plays the active role. This result is not obvious, especially when you consider caches and TLBs in your system, that might result in additional overhead because of cache and TLB misses and refill costs. This paper comes back to this in the next paragraph after the transmission for an active sender is discussed.
From the API point of view we can differentiate between three message types: single words, strings and flexpages. From the implementation point of view, another differentiation is more relevant: register versus memory messages. For register only messages it is sufficient to ensure that the data remains in the registers or is reloaded at the end. When the message descriptor or the message itself is stored in memory, page faults may happen. In this case, the kernel code has to deal with pagefaults in one of the participated user spaces.
If the pagefault happens in the sender's address space, the kernel sends it to the sender's pager. Remember, the sender was the active partner, so no additional activity is required for that. If the page fault occurs in the receiver's address space instead, the kernel has to intervene. Because of the active sender, a normally generated pagefault would be transmitted to the sender's pager. Even if the pagefault is captured and transmitted to the receiver's pager, it happens in the context of the sender. In both cases, the corresponding pagers may not reply with a mapping simply because they are not responsible for the faulting thread / address space. To solve this problem, the kernel page fault handler, when detecting a page fault in the receiver's space, temporarily switches to the receiver thread's context and invokes the pagefault again. This can for example be done through operations that do not modify the contents of the memory cell like adding a zero for write faults or reading the cell for read faults. After the pagefault is resolved, the receiver switches back to the sender that continues copying. In version X both, a pagefault in the sender's or in the receiver's context, are bound by the corresponding pagefault timeouts. The receive pagefault timeout, specified by the sender, limits the time, the receiver's pager has to handle the pagefault. The send pagefault timeout, specified by the receiver, limits the time, the sender's pager has to handle the pagefault. In Version 4, a transfer timeout limits the total time of the message transfer including pagefault handling times.
Up to now, the environment for the message transfer is described, open remains the actual way, the message is transmitted. Already mentioned was how to deal with register messages, so string copies and mappings remain.
String copies: When a string is copied by the sender, the sender's address space is active. In addition, only the kernel address space is present and accessible, because it is mapped in the upper part of each address space and the IPC is performed in supervisor mode, but not the receiver's address space [13]. The first approach would be to copy the message into a buffer in the kernel space, switch to the receiver's address space and copy the message from the buffer out to the specified destination. However this copy in copy out requires two copies and result in an additional possibility for cache misses in the in kernel buffer. To avoid this, the uniprocessor L4 μ-kernel implements the copy by temporary mapping the destination area of the receiver's address space into the kernel space. With this trick, one copy can be saved by adding additional costs for managing the

[13] Small spaces, a tagged TLB simulation on the Intel Pentium, allow part of the receiver's space to be present even if the sender's space is active. In this case, the message can directly be copied

mapping which can be neglected for longer messages. Pagefaults in that communication space are translated as if they happen in the receiver's address space like described above.

Mapping pages: Mapping a page means to copy the page table entries from the source address space's page tables to the destination one's. Since both the sender address space's and the receiver address space's page tables are present in the kernel space, a simple copy can be established. Starting from the first level page table, an entry points to the offset of a second level page table if present. If not, an empty page table may be initialized and assigned for the receivers space. Note since only present pages can be mapped, a missing page table entry of the sender, first or second level, results in an abort of the IPC.

When mapping to a destination where a page is already present, the specification tells to unmap this page, i.e. to delete the entry in all following address spaces, i.e. those where this page was mapped to. However, any currently existing implementation overwrites the location without unmapping the page.

In addition, the mapping database has to be updated with by this new mapping. Granting works like mapping except that the translations to the granted pages are deleted in the senders address space and the node in the mapping database is moved instead of creating a new child node as done for mapping.

Before starting the copy / mapping the destination area has to be checked for border violations. Remember, the receiver specifies the location where to copy / map to while the sender specifies what to transfer. If the message exceeds the receive window, the IPC is aborted with an IPC cut error.

This concludes in the following pseudo code:

```
// transmit Path if short IPC then

            // ensure the to be transfered dwords
            // are in the registers

    endif

    if long copy Message then

            while more strings to copy
            do

                map destination area to comm space
                prepare cleanup activity on interruption of the IPC

                enqueueWakeup(Sender with min(X-fer timeout
                  sender, X-fer timeout receiver))
                enable interrupts

                copy message from sender's space to comm area

                /* When switching to another thread of this task,
                 * the comm are has to be cleaned up
                 */

                disable interrupts

                if timeout occurred then

                        abortIPC ("timeout occurred")
                endif
            od
    endif

    if map Message then

            parse source page tables
            copy page entry to destination page table

            if pagefault occurred in 1.lvl ptab then

                    map free page for 2.lvl page table

            endif
    endif


    // page Faults in the Communication Space
    if pagefault occurred in Comm Space then

            store current IPC state on the stack
            switch to receiver context
            enqueueWakeup(Receiver with rcv timeout)
            unlock Receiver
            redo page fault
            if timeout occurred then

                    abortIPC ("timeout occurred")

            endif

            relock Receiver
            dequeueWakeup(Receiver)
            return to sender context
            load interrupted IPC state from the stack
    endif
```

The above explained pseudo code contains only a briefly description of the transfer path. Additional activity has to be implemented for example in the switching code, to avoid conflicts of two threads of the same task that perform long IPC and occupy the comm space. Since the cross processor IPC problem is more a problem of how to startup the IPC, this paper will not go any further into the details of this transfer path.
Up to now, we got an idea how the transfer is performed when having an active sender. The next section discusses why for uniprocessors it is symmetric whether the sender or the receiver is the active partner in the IPC. Later when considering the proxy thread implementation, we come back to this problem for cross processor ICP.

Active sender via active receiver As mentioned above, it is symmetric for uniprocessor μ-kernels whether the sender or the receiver is the active partner in an IPC. This is not obvious, especially when taking cache and TLB misses into consideration. The following paragraph tries to explain this symmetry and points out the asymmetries in the cross processor case.
In the current implementation, the sender is the active partner of the IPC. It copies the data to the receivers address space through a temporary mapping of the destination area into a communication space within the same address space. Assume the to be copied data is contiguous in the address space of the sender and no pagefaults happen in the sender's or receiver's space and no other thread is scheduled in between [14].
Copying a datastream of n bytes result in at most $n / c + 1$ cache misses – c the cacheline size – on the sender's side plus additional $n / c + 1$ on the receiver's side, remember we assumed contiguous data. The additional cache miss results from non aligned data, i.e. when the string starts in the middle of a cache line. This sums up to $2n / c + 2$ cache misses for the pure copy. Considering, the receiver will use the transferred data immediately, additional $n / c + 1$ cache misses may occur for very large n, i.e. when the receiver starts reading the copied string from the start, because during the copy, the last part displaces the first part of the string.
When switching to an active receiver, the corresponding parts of the sender's address space are mapped into the communication area that is part of the receiver's address space. After that, the receiver copies the data from the communication area into its own address space. In this scenario, up to $n / c + 1$ cache misses may occur on the sender's side as well and the same amount on the receiver's side. For large n, the additional $n / c + 1$ misses may occur as well through displacement in the caches. So in both cases, active sender or active receiver, up to $2n / c + 2$ cache misses for small and additional $n / c + 1$ for large n may occur. Assuming, the sender or the receiver accessed the location to copy from / to before, reduces the cache misses required for copying. At least for physically tagged and indexed caches like those from the Pentium processor, temporary mapping does not harm the cache situation. The temporary mapping generates an additional virtual to physical mapping to the destination data (in the active sender case and to the source data in the active receiver case). Since the cache is physically indexed [15], the data is cached at the position determined by the result of the mapping. So accessing data though different mappings does not displace the lines accessed through the former mappings. To conclude, considering cache misses, an IPC with active sender is symmetric to an active receiver IPC.

What about TLB misses? The translation lookaside buffer stores the least recently used translations from virtual to physical addresses at page granularity. When accessing a page whose entry was not in the TLB, the translation is computed and stored. Following accesses hit in the TLB until the entry is flushed or displaced by clashing ones. Copying the same n bytes as above, therefore results in at most $n / p + 1$ – p the page size – TLB - misses assuming the above described conditions of an incessant copy operation of contiguous data. Those misses occur in the sender's space when accessing the pages to copy the first time. In addition $n / p + 1$ TLB - misses are raised in the communication area in the receiver's context. So to sum up $2n / p + 2$ misses may occur in an active sender IPC. After the transfer is completed, the kernel switches to the receiver's address space requiring to flush the TLB. This leads to another $n / p + 1$ misses when accessing the copied data on receiver's side.
Assuming the sender accessed the data to copy before, reduces the TLB misses by ones the described value which leads to $2n / p + 2$ misses including the receiver side accesses to use the data.
Switching to an active receiver results in flushing the TLB on the sender's side when switching to the receiver's address space. However two times $n / p + 1$ misses may still occur. Once the misses of the communication area to read the data and second for writing to the destination in the receiver's space.
Which results in $2n / p + 2$ TLB - misses for both scenarios, actively sending or receiving.

XP message transfer In the cross processor case, this symmetry is broken up. Caches, as well as TLBs exist processor local, i.e. per processor, and cannot be used for cross processor operations. Copying data, to read it, followed by a write, need to be performed on a single processor and therefore uses the local TLBs and Caches of a single processor. One may think of parallelizing the transfer, i.e. to copy on the sender's as well as on the receiver's side. The following paragraph discusses this issue first for sender and receiver only copies.
As an immediate result, the fact that the to be copied data may be cached on the inactive thread's processor will not benefit the IPC performance. It can be even worse as concluded in section 3.2. when the to be copied data is modified. The performance results of the MESI timing section show that up to twice the cost of a miss has to be taken into consideration. So to sum up, in the above described scenario we have $n / c + 1$ cache misses for sure, though the data might be in the cache.
Considering an active sender, i.e. the message is copied on the sender's processor, $n / c + 1$ cache misses occur only if the sender's data is not cached. To this, another $n / c + 1$ misses have to be added, which probably require MESI cache coherency activity if cached on the destination processor [16]. After the transfer is completed, another $n / c + 1$ MESI including cache misses occur on the receivers side when the data is used. Compared to the uniprocessor side, we again have $2n / c + 1$ cache misses when the sender produces no cache misses, but those misses occur for large as well as smaller n and include cache coherency activity which might even double the latency.
For an active receiver IPC, the situation is better. Again we have the $n / c + 1$ cache misses for the inactive thread, i.e. the sender. Those misses occur when the receiver reads the data from the communication area and therefore include MESI activity. Compared to an active sender, that writes the destination data which are read again

[14]Those assumptions ease in the following discussion, but can be dropped easily. Non contiguous data copies can be viewed as multiple contiguous data copies. Interruptions through page faults or other threads may influence the current cache and TLB layout which can be bounded through considering multiple incessant IPCs.

[15]Remember the L1 caches of the Pentium processor are physically as well as virtually indexed because the index is extracted from the part of the address describing the page offset. The L2 Cache is both physically tagged and indexed.

[16]Cache coherency activity may occur through not participated processors and threads when the source or destination area is shared (i.e. mapped) somewhere else, too

when used, the active receiver reads those lines. Because of this, only some cachelines might lead to the costly modified to shared transition, but not all. After this cross processor read, the data is written to the destination area in the receiver's space which results in up to *n / c + 1* misses only if the buffer is not cached. MESI activity occurs only if the destination area is shared with some threads on another processor. Like in the uniprocessor case, another *n / c + 1* misses may occur for large *n* due to displacement. So to sum up, *3n / c + 3* misses may occur in total, like in the active sender scenario when the data is not cached by the sender. However only *n / c + 1* misses include MESI activity and in the case of a cached destination buffer and small *n*, only those *n / c + 1* misses occur.
As an intermediate result, the copy operation should be done on the receiver's processor.
Page table entries are cached in the local L1 and L2 caches of a processor as well and therefore are kept consistent, too. They are written to (to set the accessed bit) when a page is accessed. However despite this fact, there is no difference compared to the uniprocessor case.
Remaining is whether to use both processors, the sender's and the receiver's for the message transfer. Transferring a single string of contiguous data on two processors in parallel, requires to separate the task into two disjunct parts. Obviously copying every second byte is not such a good idea. Copying every second cacheline is a little better, but requires to synchronize for each string to copy. Copying every second string is the best to do, because each single string (portion of at most 4MB contiguous data) is the preferable granularity. For each string, the temporary mapping has to be set up anyhow, so the startup is much easier. But still we have to deal with IPC interruptions. How to signal where to restart when IPC is aborted or cut, i.e. because of a too small receive buffer. On the other hand, most messages are short, i.e. a few bytes or at most one string. When copying on one processor only, the other is free to run another thread in between. Since those overall performance benefit dominates and since long copy IPC is not such performance critical, only one processor, i.e. only one thread will actively copy. This leads to:

- ***Implementation decision 2:*** *The message is actively transferred on the receiver's processor. Only this processor is actively participated in the cross processor IPC.*

8.3.3 Problems in the Cross Processor Case

In the last section we discussed the problems that occurred in the transmit path. We concluded to transmit the message on the receivers processor, so remaining is how to startup the IPC, i.e. how to implement the send and receive paths. Before this discussion is started, this section points out the problems that occur if the uniprocessor IPC paths are used for cross processor IPC.
Design decision 1 already implied to have processor local ready and short wakeup lists to avoid the synchronization overhead for manipulating those lists cross processor. However as seen above, the uniprocessor IPC code has to manipulate those lists several times. If one of the two participants in the communication is not yet ready for the IPC, the other has to wait and therefore to be inserted into the wakeup queue. When the partner got ready for transmission, the waiting thread has to be woken up, so removed from the wakeup queue again. When the sender finished the tests, it inserts itself into the ready list. Since we decided to transmit on the receivers processor, this would be a cross processor operation, too. In an active receiving IPC, this can be avoided, since the receiver itself copies the message, however one point remain where the sender and the receiver are enqueued into the ready lists. This is when the IPC is finished. Both partners are enqueued into the ready list again to continue their work.
Despite this performance problem of enqueueing or dequeueing threads into the processor local lists, there are some more serious problems and race conditions. In the send path, before starting the IPC, the state of the receiver has to be checked, whether it is ready to receive, open or closed from the sender. If the receiver is ready, the sender starts transmitting the message. If not, it enqueues itself into the wakeup queue, waiting for the receiver to get ready or the send timeout to occur. In the uniprocessor example, those checks are done with disabled interrupts, so no other thread is able to interrupt this code. In a multiprocessor system however, immediately after the sender checked the state, the receivers state may change due to a timeout or because it just ran into the receive path becoming ready to receive now. In the first case, the sender copies the data though the receiver is no longer willing to receive and perhaps is working on the destination area. It might even be the case, that another thread takes control over the receiver's processor because we did not yet require the receiving processor to have interrupts disabled. If i.e. the receiver was openly waiting and the sender checked this state but before being able to lock the receiver, another thread interrupts the receiver and finds it openly waiting, too, the two threads start the transmission and clash in the destination area specified in the receive descriptor. This example was designed for a dual processor and becomes even worse in a n-processor system. In this case, up to n threads may concurrently send to the receiver.
This problem can be avoided on dual processor machines by disabling the interrupts on the receiver's processor before checking the state. On a Intel Pentium this can be achieved i.e. by sending an inter processor interrupt IPI and wait until the interrupt is acknowledged.
Assume, the problem is solved, as described above, for a dual processor machine. Even then there is a race condition – found by Volkmar Uhlig –, when both the sender and the receiver concurrently enters the IPC path. Assume, the sender wins the race and sends the disable interrupt signal as an IPI. However before the interrupts are disabled, the receiver enters its receive path, sending its disable interrupt signal and itself disables interrupts on its processor. The sender now detects the receiver is not yet ready to receive and inserts itself into the send path and sets himself ready to send. But immediately before this, the receiver detects the sender not to be ready yet and sets himself waiting, too. In this situation, both threads, now ready to send and receive, are waiting for the other to reach this state. Because the polling of the sender is not implemented actively, but through a check of the send queue at the receiver's side, the threads will wait until their timeouts occur or forever when specifying timeout infinity, even though the partner is ready for the communication.
To avoid this, in Volkmar's XP-IPC path, the IPC partner's TCB is locked through a spinlock before critical operations are performed. Through this, only one thread can modify the receiver, i.e. no more than one IPC will be setup to one thread and, the receiver will not change its state until the lock is released, i.e. the concurrent check of the receiver that leads to blocking both threads will not happen until the sender finished to put himself into a ready to send state. After that, the receiver continues and finds the sender in this state, setting up the IPC.
To sum up this section we can enumerate some guidelines that if not met would lead to race conditions and incorrect IPC paths.

- No more than one IPC is permitted to be set up to any one receiver at the same time.
- Only one, the sender or the receiver should setup the IPC at the same time.
- The processor local queues should be modified on that processor, only.
- Modifying the thread state cross processor requires to secure the operation through locks.

Beside this, we have to take care of an additional problem introduced by the Pentium hardware when using IPIs. The local APIC of the Pentium processor allows only to send two inter processor interrupts that are not acknowledged by the destination processor through an end of interrupt EOI signal. So if relying on the third interrupt to be delivered, i.e. in a scenario, where one IPI is spent for sending a cross processor IPC, a pagefault occurred to a pager thread on the sender's processor. In the mean time another thread performs a cross processor IPC blocking the second IPI and finally lthread_ex_regs is used to interrupt the first IPC which requires another IPI to cancel the pagefault IPC on the destination processor of the first IPC. If non of the IPIs were acknowledged, the third IPI would never go through.

8.3.4 How to trigger cross processor operations

Before actually discussing the IPC path for cross processor IPC, this section focuses on how to initiate an arbitrary operation cross processor, i.e. how processor A can invoke a method on B's processor.
Methods in general have a set of input parameters, some code that describes the desired operation and a set of return parameters. I.e. a cross processor thread_schedule call might be implemented by calling thread_schedule on the processor of the targeted thread with the same parameters like specified by the thread invoking the method. For our purpose it is sufficient to discuss only how to invoke a method with input parameters only. Output parameters can be handled as if an empty method is called on return.
Looking at the Pentium hardware, there are two possibilities to transfer information between the processors: the shared memory and the cross processor interrupts (IPIs). In the amount of transmittable data, the first one bears no strict limits, however the receiving processor needs to be triggered to know when to execute the method. The second possibility can transmit only very few amounts of data like the source processor number. But in opposite to the first triggers the operation itself. The interrupt handler can be used to start or even execute the operation itself. If only few different methods with no parameters are needed, the IPI handlers can directly be used. However at the cost of one of the 256 available interrupt vectors.
To sum up, the parameters have to be transmitted in some memory location called mailbox. With the Pentium hardware no more than one dword (4 bytes) can be modified atomically, i.e. without being interrupted. For larger amounts of parameters we have to secure the mailbox from concurrent accesses. To accomplish that the mailbox is locked in advance of the modification with a spinlock (see appendix A for a sample implementation for the Intel Pentium).
Provided we got a free mailbox and set up the message in it, we still need to trigger the destination processor to setup the operation with the parameters from the mailbox. To achieve this we can either implement polling, i.e. an active scan for filled mailboxes or send an IPI.

- **Polling:** When polling, the destination processor frequently checks the mailboxes for pending operations. The source processor sets up the mailbox by writing the parameters into the mailbox. After that, a flag is set to indicate that the destination processor should read the mailboxes contents and set up the appropriate operation.
  In the best case, immediately after the flag is set, the destination processor reads it and sets up the operation. In the worst case, the destination processor checked just before the flag was set. Then the operation is delayed until the next check. So the polling frequency bounds the setup of the cross processor operation.
  How often the destination processor can poll the mailboxes is determined by the time needed to check. If checking is too costly, polling becomes unaffordable and therefore impracticable. So how costly is checking?
  If the flags of all potential mailboxes are spread over the boxes, polling requires to read *n* cache lines for *n* mailboxes (provided the parameter size is at least one cacheline size). If packing the flags to a single cacheline, up to 32 mailboxes (1 dword) can be checked at the same time through a simple comparing by zero. In addition the remaining 7 dwords (for 32 byte cachelines) will hit in the L1 cache. So with the cost of at most one cache miss and 7 L1 hits, 32*8 = 256 mailboxes can be checked. Assuming the first access hits in the L1 cache, too, the costs for checking those 256 mailboxes sum up to 8 cycles (8 x `cmp rm32, 0`). This assumption can be met for all checks that do not result in triggering an operation, provided no clashes occur in that line. When setting the flag, the source processor writes the line which leads to an invalidation in the destination processor's cache. At the next check, the modification is detected leading to a transition from the modified on the source and invalid on the destination processor to shared on both. However a flag was set and the operation is triggered. After that the flag is reset which leads to a line in modified state on the destination and an invalid line on the source processor. Any following checks that do not detect a modification then hit in the L1 cache on the modified line. The next time, the source processor triggers another operation, the line is again transferred to the source and so on. However this is only true if the source processor only writes this line to trigger an operation and no other processor interferes.
  To sum up, despite the situation where an operation needs to be triggered, polling 256 mailboxes require 8 cycles which is neglectable.
  When checking each time the kernel mode is left, the polling frequency can be bound by the frequency of the periodic timer interrupt. In the L4 μ-kernels the timer ticks periodically every micro second. Each such tick, the timeslice of the currently running thread is decreased by one and if necessary some cleanup work in the wakeup lists is done. However a μ-kernel can be constructed such that the timer interrupt occurs at the end of the timeslice, only. The above described polling mechanism works as well for this kind of μ-kernels, despite the fact, that the polling frequency can no longer be bound by a constant timer interrupt frequency.

- **Inter Processor Interrupts:** In contrast to polling, inter processor interrupts require no additional activity on the destination processor if the operation is not triggered. IPIs are sent by writing to the Interrupt Command Register (ICR) of the local APIC. The interrupt is delivered to the destination processors specified in the upper part of the ICR. To up to 15 processors, the IPI can be delivered simultaneously by selecting the processors through a mask.
  When one of the two LINT slots LINT0 or LINT1 is available, the interrupt request is stored in the local APIC of the destination processor and the source processor can continue. The next time the interrupts were enabled on the target processor, the interrupt occurs and the handler specified by the interrupt vector is called. If both slots are occupied, the source processor blocks resending the IPI until one slot gets free. A slot has to be freed by the interrupt handler by sending an end of interrupt message EOI.
  For the purpose of triggering a cross processor operation, the interrupt handler has to check the mailboxes when activated and is responsible for the operation to be performed.
  For synchronous operations, i.e. those where the source processor has to wait actively until the operation is performed, the time to setup is determined by the time needed to transmit the interrupt request plus the execution time of the operation

and the time needed to transmit the EOI message. For asynchronous activation, i.e. those operations where the source processor can do something else in between, the time to setup the message is the time needed to write the ICR, provided a free slot exists on the destination processor. Note the operation is meant to be asynchronous for the processor only, not for the thread. The thread triggering the operation is interrupted until the operation is performed but its processor is free to schedule another thread in between. For the thread it is like if the operation is synchronously.
However compared to the polling case, there is no span when after being set up, the operation will be performed. The IPI handler that activates the operation is started exactly when the IPI is delivered (see chapter 3.3.). However provided a free slot exists, the interrupt is not delayed, so the delivery time bounds the time needed to trigger the request.
Section 3.3. also discusses the costs for the interrupt delivery on the destination processor. The most part of this time is spent for entering and exiting supervisor mode which can be saved when polling is used before leaving kernel mode.
Most of the assumptions made to determine the delivery time require a free IPI slot. To keep the slots free, even when longer operations are to be performed, the handler code should as early as possible acknowledge the interrupt by sending an EOI. Therefore it is preferable to let a thread perform the operation and use the handler only to setup this task for the thread. Those threads we call proxy threads.

Both polling and IPIs have their pros and cons: Polling has a potential higher response time and saves the additional kernel entry / exit however it is hard to bound and guarantee the response time. IPIs give this bound when handled with care, i.e. when the LINT slots were kept free. This leads us to combine both in the following algorithm:

```
// method invocation on source processor
  find_free_mailbox()
    lock_mailbox()

    setup_mailbox(parameters)

    unlock_mailbox()
    set_activate_mailbox_flag()

    **if** next_check(destination) $<$ max_latency **then**

          **if** not_yet_acknowledged **then**

                send_IPI()

          **endif**
    **endif**


  // receiver's side checking
  /* This is called at the end of the timer and by the handler*/
  check_n_ack()

    **while** mailbox_flags != 0
    **do**

      // find mailbox id
      id = bit_scan_forward(mailbox_flags)

      link_mailbox_to_proxy(mailbox[id], mailbox[id].proxy)
      set_proxy_to_running(mailbox[id].proxy)

      clear_mailbox_flag(id)

    **od**
    return


  // IPI handler
  ipi_handler()

    EOI()
    check_n_ack()
    iret
```

In the algorithm described above, the IPI is send only, when the next check of the destination processor is later than it would take to deliver the IPI. Or in general is later than a maximal latency that could be accepted. Knowing when the next check occurs however needs to have a synchronous clock of all processors and to publish the time to check next which is known for the timer interrupt, only, but not for the systemcalls that enter supervisor mode. Unfortunately on the Intel Pentium processors there is no synchronous clock available.
Note since both the check called before leaving supervisor mode and the check called by the interrupt handler run in disable interrupt mode, no conflicts occur. In the worst case, polling triggers all operation and immediately afterwards the IPIs occur finding nothing to do.

The proxy thread when activated and no operation currently in work takes the first mailbox, performs its operation and deletes this box from its queue. If the list is empty, the proxy sleeps until being woken up by the checking procedure.

Up to now we discussed in general how to trigger an operation cross processor. Now finally we continue with the XP - IPC paths in the following section.

8.3.5 The XP - IPC paths

For the ease of constructing the cross processor IPC paths, we first consider, the receiver B still waiting to receive from the sender A. A and B run on two different processors **A** and **B**. We further restrict the IPC to short register messages, i.e. no page faults may happen in the transmission.

Send Path Before trying to send, we have to check whether the receiver is present. Though we assumed the receiver to be present and waiting for the example this check is obvious so we start discussing it here. Like mentioned above, if we check for the receiver to be present on processor **A**, we cannot rely on this information. Though we found the receiving thread, it may be deleted in just this moment on some other processor including **B**. Sending the message then would find an invalid thread and therefore cannot be permitted. In this case, we have two possibilities: 1. perform any check followed by the corresponding operation on **B** or 2. do not delete the thread B without aborting A's operation.
1. requires both, the check and the delete to be performed on **B** while 2. requires to analyze A's state to know whether A is still before the check or started the IPC already. Assume for the time being, we know B is present and will not be deleted in between the IPC.
We found B, but now B has to be checked if it is ready to receive. This check leads to a similar race condition like the deletion example above. One possibility is that in the moment we found B not yet waiting for A, B enters the receive path and sets itself waiting for A. Another is that we found B ready to receive from A, but after the check, the receive timeout or an IPC abort through lthread_ex_regs aborts the IPC. Again, those races can be avoided by performing the checks on processor **B** or by delaying the timeout or abort after the check. Again assume that we found B ready to receive from A and that this will not change until the IPC happened.
We found B to be present and receiving from A, so remaining is the message transfer which is easy because no user level page faults may happen in between. The registers are copied into some kernel area (i.e. into the TCB or a mailbox) and copied out to the destination registers. By choosing the kernel stack frame as the buffer, the registers may directly be pushed onto B's stack and will be popped when returning to user mode. However after this transfer, B has to be woken up and inserted into the ready list which has to be done processor locally, i.e. on **B**.

As shown in this short (note this is only the easiest possible path for an IPC) lots of state information have to be kept, where the checks failed or were changed later on. Even worse, at the end we have to operate on processor **B** anyhow to insert B into the ready list after successfully receiving the message. So to conclude, 2. is impracticable because it makes the implementation of every other operation like lthread_ex_regs extremely hard. So how about 1. ?
Above we pointed out how to trigger an operation cross processor. As an immediate result of this section, combined with section 3.3. discussing the IPI handling times, one single longer operation bears less overhead compared to many short cross processor operations. In addition to that, performing the entire critical path of the IPC on **B** avoids the above described race conditions. This leads to:

- ***Implementation decision 3:*** *Any operation modifying a thread is only performed on its current CPU.*

So to sum up, as soon as we detect, the target B is on a different processor than A, we perform the entire IPC path on B's processor **B**. Unfortunately checking the processor of a non existing thread is impossible, so we have to check the presence of the TCB in advance. This leads to the following preliminary send path:

```
// preliminary send path 1
    if isSend() then
        from_TCB = getTCB(A)
        to_TCB = getTCB(B)

        if not exists(to_TCB) or to_TCB.myself ≠ B then
            abortIPC ("target does not exist")
        endif

        if from_TCB.cpu ≠ to_TCB.cpu then
            xp_operation: IPC Send
        endif

        // common intra processor IPC
    endif
```

At first we check whether the thread B exists by comparing the myself field of the TCB with the destination id B after checking if the TCB exists at all. If so, the current CPUs of both threads are compared and if not equal the cross processor operation: *IPC Send* is triggered. If equal, the common uniprocessor IPC path can be used, because this path is synchronized with any other cross processor operation through disabled interrupts, because we decided to allow critical cross processor operations only to be performed on the current cpu of the targeted thread.
With this trick, we reap benefit of three advantages:

1. `CLI` can be used to make the operations, though being triggered cross processor, atomic when affecting a thread. This is because any modification on that thread is done on its local processor only.
2. For intra processor IPC, most parts of the common uniprocessor IPC paths can be used.
3. The above described race conditions are automatically avoided because the operations on the targeted thread are made atomic with `CLI`

Note to point 2: Though most parts of the uniprocessor IPC paths can be used for short IPC, additional care has to be taken for long, i.e. interruptible IPC when migration is considered (see below).

Before actually discussing the XP send operation, we focus on how to setup the mailbox correctly. Implementation decision 2 concluded in transmitting the message on the receiver's processor. However due to implementation compatibility reasons with the existing uniprocessor implementation, the sender will be actively involved in the IPC. This will require only few changes in the receive path and makes the send path implementation comparable to that of the uniprocessor L4 kernel.
The sender, when detecting that the thread to send to is currently not on its cpu, sets up the XP send operation and triggers a proxy thread to handle it. Once the proxy thread gets to handle the triggered

request, it disables interrupts so no interference of other threads including the receiver may occur in between. The problematic time intervals however are after the sender started, but before the proxy thread was activated, so directly before the XP send operation and after the proxy finished its work but before the sender is reactivated again.

Let's first take a look at the first interval: If the sender is on a different CPU than the receiver, the receiver has to check the state of the sender's proxy thread instead of the sender's state. To find this proxy we store a pointer to it in the sender's thread control block. If the sender not yet accomplished this link or if the proxy is not yet started, the receiver will sleep, setting its receive timeout, waiting for being woken up by the proxy when it enters its send path. When the sender finished to set up the send proxy and this proxy starts handling the requested operation it will find the receiver ready to receive. Unfortunately if the receiver specified timeout zero, it will find the sender not ready to send and got woken up by the timeout, though the sender started its send phase already. Fortunately we can ignore this problem, because in the most common case when specifying a receive timeout zero, it wants to poll for the sender to become ready. Most often the receiver will then poll sometimes later again and then will find the sender's proxy ready to send.
We discussed how to solve the problems that the first internal bears. In the second interval there might occur a problem if the receiver immediately after it got the message starts a second receive operation from the sender. The problem occurs if the sender changes its proxy thread because it starts a second send operation, too, exactly in between the checks of the receiver. Note the receiver B has first to check whether the receiver is present, then if it is ready to send and if so if it is willing to send to B. When between two of those checks the proxy is changed, the IPC may fail or even worse, a non valid pointer to a non existent proxy may be dereferenced. To solve this, we store the pointer to the proxy in a temporary variable. All checks are performed by dereferencing this pointer which will not change in between. Additionally before returning from the XP send operation to the sender, the proxy clears the proxy pointer of the sender.

Including the above described cross processor operation startup this leads to a) the send path and b) our proxy path:

```
// send path
  if isSend() then
        from_TCB = getTCB(A)
        to_TCB = getTCB(B)

        if not exists(to_TCB) or to_TCB.myself ≠ B then
              abortIPC ("target does not exist")
        endif

        if from_TCB.cpu ≠ to_TCB.cpu then
              ipc_proxy = get_ipc_proxy
                (from_TCB.cpu, to_TCB.cpu)
              ipc_mailbox = get_ipc_mailbox
                (from_TCB.cpu, to_TCB.cpu, A, ipc_proxy)

              // write parameters for the xp send operation
              // into the mailbox
              write_ipc_data_to_mailbox

              // Set Proxy pointer
              from_TCB.proxy = ipc_proxy

              trigger_mailbox

              // set own timeout to infinity
              from_TCB.timeout = NEVER

              // Release the CPU
              from_TCB.state = locked waiting

              switchTo(Anyone)

              /* the proxy now performs the
              xp_operation: IPC Send */

              /* Here we get activated by the proxy when
              returning from the xp_operation */
              XP_RETURN

              if some error occurred then
                    returnIPC (error message)
              endif
              if not isReceive() then
                    from_TCB.state = ready
                    enqueueReadyList (A)

                    returnIPC
                      ("message successfully transmitted")
              endif
        endif

        // common intra processor IPC
  endif
```

Get_ipc_proxy and get_ipc_mailbox selects a free proxy and mailbox through an for the time being arbitrary policy (see below: Proxy Threads). The data written to the mailbox is a pointer to the sender's TCB, the destination id (B), the send descriptor and the ipc timeout of the sender. The last value has to be transmitted because we concluded from *design decision 1* to have processor local wakeup queues. Having this in mind we enqueue the proxy thread into the wakeup list in behalf of the sender if required. To do this, we have to transmit the sender's ipc timeout. Easing implementation we set the sender to timeout infinity and let the receiver if being woken up activate the sender. For the time being we postpone the wakeup of the sender from the proxy thread (see below).

```
// xp send path
    /* This path is performed by the
    proxy in the context of the sender */

    xp_send_path (from_TCB, B, snd_desc, ipc_timeout)

    // Check again if destination exists
    if not exists(to_TCB) or to_TCB.myself ≠ B then

            wakeup_sender ("target does not exist")

    endif

    from_TCB.partner = B
    proxy.partner = B

    if B is not waiting or not for me or open then

            enqueueWakeup (Proxy with A's send timeout)

            // Whom to enter into the send queue???
            to_TCB.enqueueSendQ (A or Proxy)

            proxy.state = polling

            switchTo(B)

            if returned with timeout then

                    wakeup_sender ("send timeout occurred")

            endif

            dequeueWakeup (proxy)
    endif

    // At this point, B is ready to receive from A

    proxy.state = locked running
    to_TCB.state = locked waiting
    enqueueReadyList (proxy)

    // A is now schedulable

    transmitMessage ()

    to_TCB.partner = A

    if error in the transmit phase then

            wakeup_sender ("transmission error")

    endif

    from_TCB.proxy = NULL
    wakeup_sender("Message successfully transmitted")
return to proxy
```

One open question is remaining in this path: whether to enqueue the proxy or the sender into the send queue of the receiver. The problem disappears if the proxy handles only one send operation at one time. Then it does not matter whether to activate the proxy directly in the receive path or indirectly when finding the original sender in the send queue. However to be free for further extensions of the proxy thread implementation, i.e. the proxy will perform another request while waiting for the receiver, we have to insert the sender into the send queue.

Receive Path As already mentioned above the receive path for the cross processor IPC is in most parts similar to the uniprocessor IPC path. The only two differences that we have to care about is 1) whom to wake up, the sender or its proxy and 2) which of the two to delete from the queues.

1. When the receiver, checking its send queue, respectively the specified the destination, finds the sender on a different CPU than it is, it has to check whether the sender's proxy is ready to send to us. If so it wakes it up the sender's proxy and switches to it. In the intra processor case however, the receiver has to wakeup the sender directly instead of its proxy.

2. In the intra processor case, when the uniprocessor IPC code is run, the sender is inserted and so has to be removed. In particular the sender might be in the receiver's send queue and in the wakeup queue where it has to be removed from if the receiver becomes ready to receive. Additionally, when activating the sender, it has to be inserted into the ready list.
   In the cross processor case, the proxy does the work for the sender. However the proxy enters itself to the wakeup queue, but the sender to the send queue of the receiver. So when the receiver finds a thread in its send queue that is not on its processor, it will remove it, the sender, from its send queue, but removes and activates its proxy. Activation no longer sets the sender to running, but its proxy, so the proxy has to be inserted into the ready list.

This immediately leads to the receive path for cross processor IPC:

```
// receive path if isReceive() then
    to_TCB = getTCB(B)
    if closed receive then
        from_TCB = getTCB(A)
        if from_TCB is an invalid TCB then
            abortIPC ("Invalid source")
        endif
    else
        // open
        from_TCB = B.sendQueue.dequeueHead()
    endif

    if from_TCB.cpu = to_TCB.cpu or
          from_TCB not present then
        // uniprocessor receive path
    endif

    send_proxy = from_TCB.proxy if send_proxy not present or
          send_proxy is not polling or
          send_proxy is ready to send but not to B then
        enqueueWakeup
          (B with B's receive timeout)

        to_TCB.state = waiting
        to_TCB.partner = A (nil for open receive)

        switchTo (Anyone)

        if timeout then
            abortIPC ("timeout occurred")
        endif

        dequeueWakeup (B)
    else
        // A's proxy was ready to send to B
        send_proxy.state = locked running
        enqueueReady (send_proxy)
        dequeueWakeup (send_proxy)

        B.dequeueSend (A)

        to_TCB.state = locked waiting

        switchTo (send_proxy)
    endif

    to_TCB.state = ready
    enqueueReady (B)

    returnIPC ("message successfully transmitted")
endif
```

If the thread to receive from is not yet present, no proxy threads are involved, so we can use the uniprocessor path to wait for it. When this thread is going to be created and starting a send operation to the receiver, it will wakeup the waiting receiver anyhow. Note switchTo (Anyone) lets the kernel dispatcher decide which thread to run next.

Concluding, we discussed the problems that occur when constructing a cross processor IPC path. A sample cross processor IPC path was constructed that requires a proxy thread and an ipc mailbox. Those two resources bear problems that we will discuss in the next section: Proxy Threads. Additional to that, we have one open point: How to activate the sender if the proxy is finished with its work. For the time being we assume this happens by magic somehow in the same way as the proxy was started and begin evaluating the IPC path. We will come back to this in detail in the next section.

8.4 Proxy Threads

This section looks in more detail what is happening at the proxy beside the xp send operation that it calls when being triggered to call it. The proxy thread is woken up with one or more mailbox hooked into its mailbox list. The first_mailbox pointer points to the head of this list and is used to find the current request. The last_mailbox pointer is used to more efficiently hook new mailboxes into the mailbox list and therefore points to the tail of the list.
When being activated it has to look at the first mailbox in its list and find out what operation was triggered. This can be done in three different ways: a) there is a tag in the mailbox to define the operation, b) the mailbox id itself defines the operation, i.e. there are no generic mailboxes but a set of mailboxes that are structured for a particular operation and c) the proxy id specifies the operation. Obviously b) works only for few operations and has to reserve at least one mailbox per operation but compared to a) it exactly contains the fields needed. Like b) requires a mailbox per operation, c) requires a proxy per operation. The benefit of this is however, that the proxy does not have to distinguish which operation to perform. In the current implementation the proxy thread performs at most one operation at the same time. As long as those operations are atomic, i.e. not interruptible by other threads, and the operation does not require to wait on other threads to synchronize with, the proxy is kept busy and the time it starts handling the n^{th} operation in its list is determined by the execution time of the n-1 previous operations (This assumes the proxy works without being interrupted between two operations as long as there are mailboxes in its list). However the xp send operation of the IPC has to synchronize with the receiver and if it is a long IPC it is interruptible. Both requires the proxy to wait though there might be other operations that might be executed in between.
As long as the proxy thread is busy, i.e. there are jobs pending in its mailbox, no further activation is required, however if the mailbox list is empty the system would hang if the proxy is actively polling for the mailboxes because it is still running with interrupts disabled. A far better solution would be to lay down sleeping, waiting to be reactivated through the check_n_ack operation. So the proxy would set itself to state waiting and switch to any other thread (i.e. let the dispatcher decide). The check_n_ack operation when finding the proxy waiting, it reactivates it and the proxy will go on checking its mailbox list. However if somewhere in between an ongoing operation the proxy would enter waiting state and the activation check is performed, it would get reactivated, too. This bears no problem as long as we take this fact into account and never in any proxy xp operation enter waiting state. Note the sender of the IPC (i.e. here the proxy) enters the special waiting state: polling when waiting for the receiver to get ready to receive so this bears no problems.
The remainder of this section is structured as follows: First we discuss how to wakeup the sender when the proxy finished the xp send operation. After that we look at how to implement long IPC in the proxy xp send path. Finally we discuss how to abort ongoing IPC in the cross processor situation and take a look at the problem of how to avoid that the proxy blocks waiting or some other thread to synchronize with while there are more jobs to do.

8.4.1 Wakeup of the Sender

Remembering the section about cross processor operations, we simplified the model of calling a cross processor operation by dividing it into two single path operations. A invokes a cross processor operation with the input parameters taking B as its proxy and B replies with an empty cross processor operation taking A as its proxy and transmitting the output parameters in the mailbox involved.
The same trick can be used for reactivating the sender. We define the sender to be the receiving proxy of a cross processor operation. The sender when activated reads the output parameters which happen to be none and frees the mailbox it receives the cross processor operation in.
With this in mind we can fit in the open operations into the IPC paths:

```
//wakeup_sender
    wakeup_sender (error code)

    // set error code
    from_TCB.xp_cc = error code;

    ipc_mailbox = unhook first mailbox from list
    from_TCB.proxy = NULL

    setup ipc_mailbox with proxy = A

    trigger_mailbox

return to proxy at end of IPC


    // XP_RETURN
    free and release all mailboxes in the sender's mailbox list
```

Wakeup sender writes the error code to the senders TCB and sets up a mailbox and triggers it. Since the ipc_mailbox previously used to trigger the xp send operation is no longer needed, it can be used for the reply. In the uniprocessor send phase, abort IPC respectively return IPC leaves the kernel directly. However wakeup_sender has to return to the end of the IPC path because the proxy has to do some cleanup work.

8.4.2 How to deal with page faults in the long IPC

A long copy IPC is specified through a message dope in the address space of the sender. It no longer restricts the message to copy to few registers, but requires to copy some strings from the sender's address space into the receiver's address space. As the sender does, the receiver specifies the location where to copy to through a message dope in memory.
So what might happen is that we get a pagefault in one of those four regions, the sender's message dope, the receiver's message dope, the location from where to copy in the senders address space and the location where to copy to in the receiver's address space. As mentioned in the introductory sections the destination location is mapped to a comm area that is part of the kernel space. So what to do: If there is a pagefault in the sender's context we can directly generate a pagefault IPC. If there is a pagefault in the copy area

we first have to translate it to the corresponding location in the receiver's space and setup the operation for the receiver.
How does this change if a send proxy is included? Before starting the copy, the proxy switches the user space (Note the proxy itself has no address space but runs inside the kernel, only) to the sender's space. When starting the copy and there is a pagefault on the receiver's side, it is translated to the receiver's context like it is done in the intra (uni-) processor case. However instead of the sender faulting directly, the proxy thread faults in behalf of the sender. In this case we have to setup the pagefault IPC to the senders pager as if the sender did fault directly and if the sender receives the mapping (i.e. the reply to the pagefault) reactivate the proxy. Unfortunately if a proxy is involved, the sender is on a different processor.
To solve this, the proxy has to restart the sender, not with an empty operation, but with a request for redoing this pagefault. The sender when being activated generates a pagefault in its address space of the appropriate type by touching the same location. After receiving the mapping, the sender reactivates the proxy at the interrupted position. Unfortunately this takes quite some time.

8.4.3 How to abort ongoing IPCs

When calling lthread_ex_regs, any ongoing IPC from or to the targeted thread is aborted. If a pagefault IPC is nested, i.e. because of a pagefault that occurred during a long IPC, both, the nested pagefault and the previous IPC have to be aborted.
In a single uniprocessor IPC at most three threads are involved: the sender (A), the receiver (B) and if a pagefault happened in between one of the two thread's pager (P). In the cross processor IPC case, those three threads are on three different processors which means that additionally up to two proxies are involved: the sender's proxy (S.X) and one additional proxy for sending the page fault IPC or the mapping replied (P.X).
In general a fourth processor is involved, that of the thread (L) calling lthread_ex_regs requesting to abort the ongoing IPC. What we have is a cross processor operation, but instead of two processors being involved, up to four might be involved depending on the states of the three other threads. Assuming we have the worst situation, where all four threads are on different processors, then L triggers a proxy L.P1 on the destination processor which is one of the three threads A, B or P. This proxy checks the state and initiates a second and if a pagefault happened a third proxy. Once all those proxies were activated, the IPC can be aborted. As long as the IPC proxies may block, we need to have a second type of proxy handling IPC unwind operations. To ease implementation we take the same mailbox for all those (up to three) proxies, collecting the state of the participated threads and unwinding the IPCs of all the participated threads local to that proxy. The following picture shows the possible state of all the participated threads. Each unwind proxy has to find out in which state the threads that are locally participated are and after collecting this, unwind the IPC. Note when those proxies are all activated, the state is no more changed because all proxies disable the interrupts. Because of we are using the same mailbox for all proxies, no additional proxy has to be activated if two or more threads are on the same processor. The proxy will automatically find it in the shared mailbox though not triggered explicitely.
Two or more concurrent lthread_ex_regs operations may be performed concurrently, as long as the proxy thread detects the state of the thread whose IPC it eventually has to abort, sets up the dependent operations and after finishing the abort operation handles the next operation. This of course assumes that in between the participated threads are not migrated. This i.e. can be achieved through aquireing a "do not migrate" spin lock (see below).

8.4.4 How to avoid blocking the Proxy on XP IPC

As mentioned above, the Proxy may block on non atomic operations or on operations that require synchronization. IPC does both. The obvious first try is to construct the proxy such, that if it blocks it continues with another operation. Assuming there are sufficient many mailboxes in the system, we may continue with the next job. However in this case, more than one operation is currently in processing and we have to keep track of all. To do this we may remove the mailbox from the proxy's mailbox list and insert it to some datastructure that keeps currently in work mailboxes. Instead of the first mailbox, we include a pointer indicating the operation currently in work. Remaining is the problem of how to reactivate an interrupted operation. To solve this, we have, when we block on the current operation or when the operation is finished, to check the pending operations if one of them can be continued before accepting the next operation. This again bounds the startup time of the n^{th} operation to the sum of the handling time of the previous n-1 operations plus some management overhead for the pending operations.
The details, how to organize this data structure, how to construct the proxy operations to be easily restartable and how to optimize the polling for a change of the pending operations open a large playground for implementation tricks, but those are beyond the scope of this thesis so we will not go any further into detail here and continue with another approach.

For short, fast operations, proxy threads are preferable, because they need just to be triggered and perform the desired operation in behalf of the triggering thread. Another possibility however to perform an operation cross processor is to migrate to this processor, perform the operation and finally migrate back. Unfortunately migrating is costly and involves a request to a user level scheduler if it is not hidden in the kernel implementation. Of course migration itself has to involve a proxy thread on the destination processor. So what if we use this trick to block the sender directly instead of its proxy.
For ease of discussion we restrict ourselfs to cross processor IPC for the following paragraph. When the proxy has to block it migrates the sender to its processor and lets the sender perform the appropriate operation itself. But before returning to user level, it has to migrate itself back to its originating processor. This can be achieved by inserting an artificial frame on the sender's kernel stack that when leaving the IPC operation returns to some special code that migrates the thread back. Of course for any user thread, the sender should remain on its current processor and the migration should be transparently performed.
With this trick, we can handle pagefaults in long IPC more efficiently because now, the sender directly faults and if reactivated because of receiving the mapping from its pager, the sender may directly continue.
If we expect that long IPC will probably generate a pagefault, we can directly migrate the sender instead of involving a proxy thread. As an immediate result, the proxy is busy all the time when an operation is pending.

8.5 Races due to migration

The above described IPC paths implicitly require that none of the participated threads are migrated after one of the threads entered its IPC path until the IPC is finished. If this is not met, there might occur three race conditions for short IPC:

1. the sender is migrated in between the proxy thread reads its CPU and the activating IPI reaches the sender. The IPI will

be delivered to the destination processor, however the sender will no longer be there.

2. The second race occurs if after the sender checks the CPU of the receiver and before the proxy is started, the receiver is migrated to another CPU. Those two race conditions have in common, that the target is migrated in between the delivery time of the cross processor operation. Once the proxy is started, it runs in interrupts disabled and no migration request may be performed in between.

3. Unfortunately there is a third race condition if the sender sets up its proxy, this proxy finds the receiver not yet ready to receive and inserts the sender into the receiver's send queue and blocks. At this points the receiver is migrated before it is able to enter the receive path. The receiver might be migrated to a third CPU, i.e. the sender, the receiver and the sender's proxy are on different processors. In this case, we have to include a proxy thread on the receiver's proxy and abort the operation of the current proxy. But what if the receiver is migrated to the sender's processor. In this case we can restart the IPC locally for both threads, the sender and the receiver as a simple intra processor IPC.

As seen above there are many operations to distinguish for the cross processor IPC path even worse, we have to take into account all these conditions into the lthread_ex_regs path when aborting the IPC. To circumvent this, we have to reduce the possible states.
To avoid the first kind of race conditions, we can spin lock the targeted thread, so enforcing the migration request to aquire the lock before. For the sender, the same can be reached if we do not migrate the sender while a proxy thread handles a request for it.
The second kind of race condition can be avoided by when migrating the receiver, we check its send queue for threads whose proxies are still activated and migrate those requests with the thread. Pending requests, i.e. those that did not yet activate a proxy thread we will find by calling check_n_ack before and migrating all the requests to that thread until no request is pending.
Both methods are at the cost of delaying or blocking the migration request. Migration however is used by the scheduling server or more precisely by the load balancing server to migrate threads to adapt to unbalanced loads. Blocking the scheduler to long might then affect the entire schedule.
Lazy migration, i.e. the receiver sets up a request for migration and the kernel hooks this request to some proxy but in between return back to the scheduler (i.e. the thread calling thread_schedule) immediately and performing the request sometimes later. This would solve the problem of delaying the actual migration to a safe point, but future work about cross processor scheduling which was not covered in this thesis has to show whether this approach pays. This however, including to find the right abstractions for scheduling is far beyond the scope of this thesis.

8.6 Thread Control Block structure

Looking at the IPC path, we can divide the entries in a thread's TCB into three different groups:

1. fields that are used processor local only,

2. fields that are frequently written cross processor and

3. fields that are read only.

3) also includes fields that are modified very infrequently but read more often.
From section 3.1. can be easily concluded that no two fields of different groups should be in the same cacheline. Having this in mind we just have to identify and rearrange the fields of the TCB appropriately.

1. group: local fields:
   (a) **ipc_buffer:** a buffer used to temporarily store the register dwords in the C++ kernel
   (b) **ipc_timeout:** the send / receive timeout of the thread
   (c) **pager:** the thread to deliver this thread's pagefaults to
   (d) **stack:** when switching to another thread, the stack pointer is stored in here
   (e) **thread_state:** the state of the thread, i.e. running, waiting, polling
   (f) **priority:** the priority of this thread
   (g) **timeslice:** the total timeslice
   (h) **remaining timeslice:** the remaining part of the total timeslice
   (i) **link pointers for the ready, wakeup**
   (j) **resources:** the resources held by this thread
   (k) **comm area:** which part is mapped into the comm area
   (l) **link pointers for the send queue**
2. group: global modified fields:
   (a) **partner** the partner to communicate to (the proxy sets this field for the sender and the receiver)
   (b) **proxy** pointer to the proxies TCB
   (c) **xp_cc** the completion code of a cross processor IPC
   (d) **pointer to first and last mailbox** if the thread is a proxy (or a sender see above)
3. group: global read only fields:
   (a) **myself:** Thread id of the thread represented by this TCB
   (b) **cpu:** The cpu, this thread is currently assigned to. This field is assumed read only because migration should happen much less frequently, than this field is checked.
   (c) **link pointers for the present list**

This leads to the following possible structure of the TCB for 32 byte cachelines like they are in the Pentium processors [17]. The TCB structure is noted in C-style and is derived from types of the L4-ka Hazelnut kernel. In brackets after the name, the size of each element in bytes is denoted.
As seen above, the TCB needs only one global cacheline that can be assumed to be in shared state because it is modified infrequently and one line for that is modified more often. However we have to spent three local lines, so how to arrange the fields above. If processor local IPC should be most efficient we can optimize this by analyzing the sequence of accesses to the TCB fields that occurs in the "fast" IPC path, i.e. for short IPC where the receiver is willing to send while the receiver is still waiting. As can be seen above, I arranged the fields of the tcb such, that the first line covers

[17]Note P4 has 64 byte wide cachelines

Type	Field	Size
Processor local cachelines		
dword_t	ipc_buffer[3];	(12)
timeout_t	ipc_timeout;	(4)
tcb_t	*ready_prev;	(4)
tcb_t	*ready_next;	(4)
dword_t	thread_state;	(4)
ptr_t	stack;	(4)
		—
tcb_t	*wakeup_prev;	(4)
tcb_t	*wakeup_next;	(4)
dword_t	resources;	(4)
dword_t	comm_area1;	(4)
dword_t	comm_area2;	(4)
tcb_t	*send_queue;	(4)
tcb_t	*send_prev;	(4)
tcb_t	*send_next;	(4)
		—
dword_t	priority;	(4)
sdword_t	timeslice;	(4)
sdword_t	remaining_timeslice;	(4)
l4_threadid_t	pager;	(4)
	DUMMY FIELDS	(16)
		—
global shared cacheline		
l4_threadid_t	myself;	(4)
dword_t	cpu;	(4)
tcb_t	*present_prev;	(4)
tcb_t	*present_next;	(4)
	DUMMY FIELDS	(16)
		—
global modified cachelines		
l4_threadid_t	partner;	(4)
tcb_t	*proxy;	(4)
dword_t	xp_cc;	(4)
mailbox_template_t	*first_mailbox;	(4)
mailbox_template_t	*last_mailbox;	(4)
	DUMMY FIELDS	(12)
		—

Table 3: Sample TCB structure for 32 byte cachelines (— denotes end of cacheline)

the fields needed for short IPC like the message buffer, the stack that is read when entering the kernel, and the thread state. The second line holds the remaining fields participated in IPC: the send queue, the resources including the comm area and the wakeup list. The third line covers the remaining fields for scheduling and the thread's pager. The ready list is put in the first line too, to fill up the remaining space.

8.7 Migration

Migration moves a thread t_1 from a source processor B to a destination processor C. *Design decisions 2 and 3* concluded in extending thread_schedule to migrate threads explicitly. We already rejected implicit migration i.e. on IPC or due to fine grain scheduling in the design decision process described above.

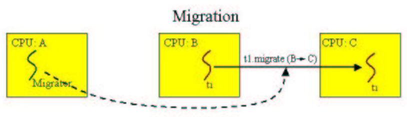

Figure 7: Thread migration

8.7.1 Thread migration

What is required to migrate a thread? First an algorithm for migrating ready to run threads will be explained and discussed. Later on this algorithm is extended for waiting threads and threads that are migrated within the IPC message transfer.
From the design section we concluded to have processor local ready lists. The main reason for this was because they require no synchronization when accessed locally only. Transferring a ready to run thread however has to remove and reinsert the migrated thread from B's and into C's ready lists. In the general case, the migration request can be initiated by a thread on a third processor A which complicates the implementation, because then both, deletion and insertion is a cross processor initiated operation.
Despite managing the ready lists, the migration code has to free all locally held resources, before the transfer can be completed. As described in the introductory section 2.4., the Intel Pentium processor allows to delay the saving of the floating point registers until used by a different thread. However, since the FPUs are processor local and since we are migrating a thread which might have accessed the FPU before, the floating point registers have to be saved and the FPU be released before migration. Note, the storing can't be delayed until the to be migrated thread t_1 uses it again, because t_1 can be scheduled on the new processor without notification to the old one. Remember, there is no easy and fast way to trigger a processor local operation cross processor, i.e. to store the registers of B, when C's FPU is used next. Similar actions like for storing the FPU registers are required for the remaining resources.
The last thing to do is to transfer the TCB which results because of the MESI cache coherence protocol and the shared memory architecture in changing the current processor number, only. This is possible because the TCBs are mapped to each processors kernel space (see below) so they are visible from any processor in the system.
The critical point in the above explained algorithm is the deletion and insertion into the ready lists. If not done processor locally, this would require an atomic operation to do the job or synchronization whenever the lists are accessed, even for the currently executing thread. Some processors offer atomic operations to compare and exchange two times two values in parallel. The Intel Pentium does not. Since both, a two to two compare is not available and synchronization is not acceptable, the operation has to be invoked locally.

8.7.2 A Migration algorithm

The algorithm is divided into three parts, where **A** is run on processor A, i.e. where thread_schedule is called, **B** on the source processor B and **C** on the target CPU C.

Notation: (= x) indicates the xth synchronization barrier.

```
A
    if destination thread $t_1$ needs to be migrated then
        trigger migration task B and C
    endif
    (= 1)
    exchange the timeslice and priority
    (= 2)
    return to the invoking scheduling sever.

    B
    enter kernel mode in B
    (= 1)
    delete $t_1$ from local ready list
    free any resources held by $t_1$
    (= 2)
    set new processor number
    (  3)
    if $t_1$ was currently running then
        switch to next thread
    else
        return to currently running thread
    endif

    C
    enter kernel mode in C
    (= 2)
    insert $t_1$ into local ready list
    (3̃)
    if $t_1$ priority is higher then
        switch to $t_1$
    else
        return to currently running thread
    endif
```

When calling thread_schedule, the processor A enters kernel mode, because the TLBs have to be modified. Like in the uniprocessor case, the call parses the arguments, selects the targeted thread and checks whether modifications are allowed. Depending on the arguments in the registers, the kernel code exchanges the corresponding TLB entries with the new values. The old values are prepared for the output parameters. In the algorithm, exchange timeslice and priority stands for that. Of course, since triggering cross-processor operations takes some time, it is preferable to initiate the request as early as possible. In fact this can be done directly after finding the thread, checking whether modifications are allowed and that the thread has to be migrated. In the cross-processor case, when A and B differs, we have to wait until B enters kernel mode,

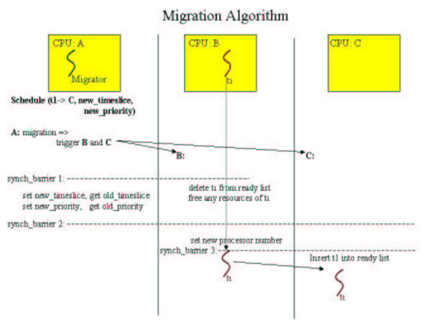
Migration Algorithm
CPU: A
Migrator
CPU: B
t1
CPU: C
Schedule (t1-> C, new_timeslice, new_priority)
A: migration => trigger B and C
B:
C:
synch_barrier 1:
set new_timeslice, get old_timeslice
set new_priority, get old_priority
delete t1 from ready list
free any resources of t1
synch_barrier 2:
set new processor number
synch_barrier 3:
Insert t1 into ready list
t1
t1

Figure 8: The thread migration algorithm

i.e. until the invariant, described above takes into account, because otherwise this thread might be scheduled on B with inconsistent values.

The ready lists are organized as a connection of TLBs of threads with the same processor. In particular, a processor local array contains a head pointer per priority. This head may point to a TCB of the same priority which might itself be connected with the next TCB. Since within the same priority, round robin is implemented, a double linked list is used to find the next (i.e. where the head points to) and the tail of the list (the backward pointer of the next thread). Due to this header array, the dispatcher has to parse in the worst case the entire array for a head pointer to a valid thread. As an immediate consequence, changing the priority of a thread requires to delete the thread from the old priorities list and reinsert it to the new one, which is nothing more than executing first **B** and then **C** on the same processor. When migrating, the processor B and C differs.

B deletes the thread from its current ready list, frees the resources held by t_1 and C reinserts it.

Obviously we should not insert a thread into the new queue before deleting it from the old. But this is not true. If we know that the B and C are in kernel mode with interrupts disabled, no thread is actively running, so especially t_1 cannot be scheduled in an inconsistent state. To ensure this precondition, we have to explicitly synchronize the three methods, A, B and C after entering kernel mode. At that point, the insertion and deletion can be performed and the scheduling parameters can be modified. To avoid on the other hand, that one of the three methods return, before the operation is complete, a second synchronization is needed afterwards.

In detail, the situation is as follows. To exchange the scheduling parameters, A has to ensure that B is in kernel mode, i.e. no thread is running on B. This is done by the barrier synchronization point (= 1). For the rest, of the operation B is independent from A and C and might continue, even return back to user mode. t_1 is no longer present on B and will not be found locally, if a concurrent scheduling request is performed. This is because B changed the processor number before returning from the cross-processor operation so by now, t_1 is on the target processor.

C instead can start immediately after the scheduling parameters are written, because the new priority need to be known to insert the thread into the right priority ready list. This is ensured through the synchronization barrier (= 2).

At this point A gets independent of C and may continue to user level, provided no acknowledgment of the operation is required. But since the receive of the IPI or the request in a mailbox was acknowledged by (= 1) and (= 2), the operation will not fail provided implemented correctly. C inserts the thread into the ready queue and has to decide, whether to directly switch to it, if t_1 is now the highest priority thread or to the last thread, that had the highest priority before.

The additional synchronization point (= 2) in B has to be inserted when using a single link location in the TCB for the ready list. Then the link has first to be freed by the deletion process, before it can be used for reinsertion. Otherwise the following threads in the list are lost, when the pointers to them are overwritten by the new values of the destination.
The last synchronization point ($\tilde{3}$) is not a strict synchronization, but requires weaker condition only(denoted by $\tilde{}$ instead of =). For the above described synchronization points it was always required that both reach the synch barrier before any of the two may continue. However ($\tilde{3}$) requires only C to wait until B has set the processor number, but B may already switch. If C would not wait, the next operation on t_1 might still find the old processor number which might lead to wrong assumptions and unnecessary cross processor operations or even lead to fail the operation.
Normally, a race condition occurs when immediately after B finished, a second thread_schedule system call is initiated on t_1. However since the operations A, B and C are performed atomically in interrupt disable mode, the insertion operation on C would first be completed before any other modification to the ready lists can be performed, so no race condition occurs.
For the two special cases A == B or A == C, A has to be performed first. The remaining synch points with B respectively C can be ignored and of course no IPI has to be send.

8.7.3 Waiting threads

In contrast to running threads, waiting threads are additionally enqueued into a waiting queue with a certain timeout. Note when setting a thread to wait, it is not explicitly deleted from the ready queue, but the dispatcher checks the queue for not ready threads and skips those. This implementation trick is an optimization for threads that are frequently waiting for some one, but only for a short time. It already was built in the uniprocessor μ-kernel by Jochen Liedtke.
So additionally to deleting and reinserting a thread from the ready lists, it has to be deleted and if the timeout did not expire in between reinserted into the destination processors wakeup queues. Threads waiting with an infinitely large timeout are not inserted into the wakeup lists.
Migrating waiting threads does not evolve new problems compared to migrating ready threads, however migrating threads that are within an IPC may cause some problems (see below).

8.7.4 Migrating the Migrator

Migrating the migrator itself causes serious problems. When the scheduling server wants to migrate itself, the algorithm can be executed like for any other thread, however when actually switching back to user level the kernel stack of this thread is compromised by the remaining part of the migration code on the other processor.
However it can be assumed that in a multiprocessor operating system, a scheduling server exists per processor because of performance reasons. Otherwise any scheduling would crosses the processor boundaries and is thus far to slow.
Restricting migration such, that the originator cannot migrate itself does not harm. On the other hand, if needed, it can be implemented in user level easily by taking two threads a scheduling server thread and a scheduling server migrate thread. When the first wants to be migrated, it sends the request to the migrate thread and afterwards pulls this thread to its new processor. This leads to:

- ***Implementation decision 4:*** *No thread can migrate itself. The request will simply be ignored by the μ-kernel.*

8.8 Kernel Address Spaces

In L4, the kernel address space, including all local data structures and the physical memory is mapped to the upper 512 MBs of each task. This allows to efficiently call the kernel functions i.e. the system calls, because the intermediate switch to a kernel address space can be avoided. When switching to another thread in a different address space, the new page directory is loaded (and the TLB is flushed and refilled), but no intermediate loading of a kernel page directory is required.
In a message based or NUMA multiprocessor, local data is kept in the local memory modules that are within the same node. In SMPs, all memory modules are processor local, so in which part of the physical memory the local kernel data structures are stored does not matter.

8.8.1 How to find processor local data?

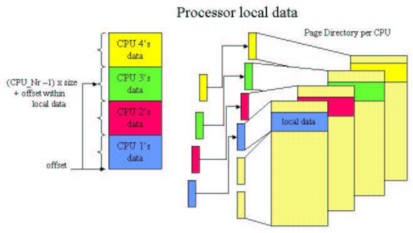

Figure 9: Processor local data implementation

Usually each processor of the SMP has some specified register or memory mapped hardware like the local APIC. Those locations (we use the Task Register TR and the memory mapped APIC_ID register) are used to store the current processor number. Given this number, the local data structures can be found by storing them continuously in memory and adding the processor number times the size of the data structure to a fixed offset to get the i^{th} processor's data.
As mentioned above, we use the Task Register of the Intel Pentium to store the processor number. However because of the build in operating system of the Intel Architecture, the register can be loaded with a valid task state segment descriptor only. Because of this, we built a temporary Gate Descriptor Table (GDT) containing a task state segment per processor and switch to that GDT temporarily for loading the processor number. To get this number, the processors spin on a processor field in memory with an atomic exchange and add instruction, increasing the processor field and storing the old value as processor number. When switching back to the system's GDT, the task register remains valid and is never checked against the task state segment descriptors again, provided it is never written thereafter. Note loading the task register, which can be done with `LTR`, can be done in supervisor mode only. Reading the task register can be done in roughly 10 cycles with the `STR` instruction in supervisor as well as in user mode.
Though the processor number can be found efficiently, each access

to local data requires to compute the offset address first. To avoid this, two additional tricks can be applied. First, execute different code, in which the pointer addresses have been computed at compile time. This requires additional space to store the different code and to know the maximum available processor number in advance. The second trick, that we use requires no modification to the accessing code at all. The MMU and the TLBs are used to translate a virtual address to the corresponding physical address. This can be used to translate the same virtual address on different processors to different physical addresses. However this restricts the granularity of local data to memory pages and requires a first level page table per processor per task. With this implementation trick, the processor local kernel tables are mapped to the same virtual addresses on all processors and the MMU scatters them to different physical pages.
Additionally to the simple implementation of local kernel tables, three other tricks come for free: local long IPC communication spaces, a local dispatcher thread and local proxy threads. However remember, *p-1* additional pages are required per task where *p* is the number of CPUs in the SMPs. Second, this additional page tables have to be synchronized [18].

8.8.2 Long IPC Com Spaces

When copying data between two address spaces, the IPC code first has to copy the data into the kernel area, switch to the destination task and copy the data back to user space, because no direct copy to the destination address space is possible. However, this method is inperformant because two copies are needed which will for sure miss in the kernel area and additional switches are required, if the message size exceeds the buffer size. To avoid this, the uniprocessor L4 μ-kernel temporarily maps the pages of the destination task into a kernel area called com space. Since at most 4MB portions can be specified and copied as a single string, 2x 4MB are mapped to avoid border checking [19]. In the SMP kernel we have to reserve an area per processor to map the copy destination, since we do not want to restrict long IPC such, that only one copy operation may be performed at the same time. However with the per processor per task page directory trick, we get the local mapping area for free.

8.8.3 Processor Local Dispatcher Threads

Design decision 1 requires that the dispatcher thread and the ready queues are processor local. Though the L4 dispatcher thread needs no state information (this means no TCB is required to store its state on a switch), it needs access to the ready queues which are realized as an array of priorities, holding the head pointers to the next ready TCB of that priority. The TCBs itself are connected in a double linked list. Since we decided that this information is accessible processor local only (*design decision 1*), it can be kept in a local page.
This automatically avoids a cross-processor access to those tables. Remember, allowing cross-processor accesses would lead to unacceptable overhead for synchronization.

8.8.4 Processor Local Proxy Threads

The proxy thread concept is a trick to perform cross-processor operations like IPC locally. The cross-processor IPC request is transmitted to a proxy thread through a mailbox. Then, this thread sets up a normal intra processor IPC to the original receiver (see above). Since proxy threads exist processor local only they can be stored in a local page.
A thread is contacted through its thread_id. This id is unique and globally visible in the system. Unfortunately there exists a direct relationship between the thread_id and the position of the TCB in the current kernel, which is permanently used to find a thread's TCB by masking out the version and chief (a relict from Version 2, see Version 2 spec) bits and adding the offset of the TCB area to the thread_id. So when sending or receiving a regular IPC to or from a proxy thread, this thread_id is parsed and since the TCBs are mapped locally only, the local proxy thread is reached.
In other words, another processor's proxy threads are invisible and can't be accessed, except through its mailbox (which better should not be in a local page).

8.9 Superfast IPC

Super fast IPC (LIPC) is an implementation of some flavors of IPC in user level. Avoiding the kernel entry and exit restricts this IPC to very few cases. On the other hand, it makes those cases really fast. Implementing an IPC path in user level requires access to some parts of the thread control block (TCB) in user level. These parts are stored in the UTCB.
Since cross-processor IPC requires kernel intervention, LIPC will setup a normal, in kernel IPC in the cross processor case. LIPC is for intra task communication only, which means that the UTCB is readable and especially the current processor field of the UTCB. Cross-processor IPC will be detected and if specified aborted before entering the kernel.

At this point the work done in this study theses ends. The remaining section describes some open questions that came up during my work and are not handled in this thesis. Afterwards this paper is concluded.

9 Open Questions and Future work

This section refers to open questions and not yet evaluated ideas and optimizations for the L4 SMP kernel. Two additional sections are included: scheduling cache colors and memory accesses, that refer to papers and further research topics mentioned in the thesis.

9.1 Covert Channels

In the days of Internet, hundreds of applications are downloaded and executed, i.e. to mobile devices, from a more or less trusted source. The communication network itself, that spans the Internet is a huge playground for attackers. In order to secure the private systems several security applications and hardware were built, i.e. firewalls, crypto hardware and so on.
To sum up, there is a need for a monolithic operating system as well as for a μ-kernel to support some mechanism to protect and secure the system against attacks or involuntary accesses from outside. Unix establishes a user / group security policy using passwords to authenticate. But that kind of security policies is not what we will discuss next.
Assume an application is downloaded i.e. a game that frequently has to synchronize with some server on the Internet and therefore has to be allowed to access the net. Another application runs in background that manages your bank account data, that is offline available. Since the management software was not provided by your bank, i.e. a shareware program, you do not want it to publish your bank data over the Internet.
In this scenario one application, the game, needs to access the Internet while the other is not allowed to. Operating system security

[18]E70: Some Pentium processors may deadlock if the access and dirty bits of two page directory entries pointing to the same page table are allowed to become inconsistent. The μ-kernel has to ensure the consistency of those two bits.

[19]Note 4MB is the area that is covered by an entire 2. level page table, so mapping requires only to copy the two page directory entries.

and user level software like firewalls ensure that the network stack is accessible for the game only and protection like the address space plus some additional IPC controller encapsulates the game from the management software.
Though there is no mean provided by the operating system to communicate, the two applications may exchange data. Of course both applications need to access their own data on the disk for example. This shared resource can be used to transmit the bank account information to the game and out through the Internet. Since only private files can be accessed, there is no way to communicate directly, but by accessing the disk in a certain pattern, generates different response times for others. When the filesystem is servicing a job of the management software, the concurrent access of the game is delayed. The game can now poll frequently the disk and measure its service time by reading the clock or generating a clock thread that increases a counter. This communication line is an example of a covert channel.
This example hopefully showed two things: a) there are many possible ways to construct covert channels. Not all are such obviously, and b) there is no really efficient way to get rid of this channel, because delaying the filesystem request for a changing period, which would introduce noise to the channel, drastically reduces the response time guarantees a filesystem can give.
Covert channel are an open problem and will be taken into account for future μ-kernel design but not in this thesis. However we will revisit the design, especially cross-processor IPC detection when covert channels are examined.

9.2 Local Small Spaces

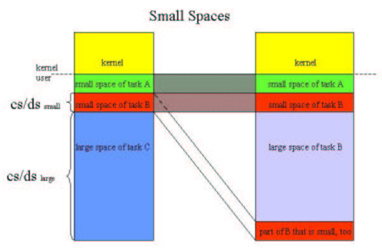

Figure 10: Small spaces

Small Spaces are a software simulation of a tagged TLB on the Intel x86 architecture. A tagged translation lookaside buffer caches the translations of virtual to physical addresses like an untagged TLB does. In addition to the virtual address, a task number is stored and compared when accessing data from the caches. Untagged TLBs have to be flushed on a task switch, because the same virtual address may point to different physical addresses. Tagged TLBs do not have this problem, because the address space number is part of the cached translation and is considered in the compare. This means though having the same virtual address, the translations differ when the address space numbers differ.
Small spaces simulate tagged TLBs. The first part of the user address space is mapped to and shared among all tasks, like the kernel is with the difference, that the kernel pages are not user accessible. The segments of the Intel architecture are used to protect these pages. Segmentation introduces another level of indirection in the address translation. Before getting the virtual address, the segment base address is added and the corresponding address is compared against the segment limit. Segment border faults raise an exception, that can be handled by the kernel. This can be used to transparently create a small space and relocate is behind the big address space (after 3GB). When the segment limit is violated, the μ-kernel switches to the corresponding large address space.
When switching to a small space, the segment descriptors are changed to point to a part above 3GB, where the pages of the task are mapped to. Switching to another small space resets the segment descriptors again to cover that space's data pages. Only when switching back to another large space, the page directory is reloaded and the TLB has to be flushed. Since the segment mechanism automatically adds the segment offset to the virtual addresses, small spaces are transparent for the user. When accessing data beyond the segment limit, the exception indicating a segment violation is caught by the kernel which switches to the large space of that task and executes the faulting instruction again. Page faults are translated as if they happen in the large space and transmitted to the faulting thread's pager. The decision, which task to run in a small space, can be set by a user level scheduling server through the μ-kernel call thread_schedule.

The untagged TLBs of the Pentium processors exist locally, which leads to the idea of simulating a local tagged TLB. Local small spaces work like small spaces in the L4 uniprocessor kernel, with the difference, that the mapping, tasks to small space can differ. So on processor 1 task A can be in the 2nd small space, while on processor 2, the same task can be in small space 1 or even in a large space.
Assuming a per processor, per task page table seems to limit the implementation problem, because the processor tables holding the segment descriptors (the Gate Descriptor Table GDT) is processor local. Potential problems may occur if page faults are raised concurrently once in a small and at the same time in a large space, but this is a similar problem as if concurrent page faults are raised in the same large space. In such a case, the μ-kernel will deliver two page faults, because the page is accessed by two different threads and a user level pager should reply with mapping a page or resuming the thread by replying with a zero IPC. Note the same scenario may occur on a uniprocessor if the pager is not in receive state. Thread t_1 touches the page, a PF-IPC is set up, thread t_2 is scheduled, because the pager is not waiting for the IPC yet, a second PF-IPC is set up, the pager enters receive state and finds two PF-IPCs for the same page from two different threads. A second problem is how to specify which thread will run in which small space. Currently the *thread_schedule* systemcall is used to determine the task's small space. Either this call is defined to work locally only, or a destination processor or a list has to be defined to specify the small space processor pair.
Care has to be taken when an IPC is set up between two small spaces, that crosses processor boundaries. The problem is that though a direct small to small IPC can be set up for one processor, this might not be the case on the other processor, i.e. because the small space on the sending processor A might be large on the receiver's processor B. To solve this problem, the μ-kernel has to check what transfer mode is possible on the processor where the transfer happens. In the worst case, a temporary mapping of the missing space's portion has to be established like for large to large long IPC.
Future work has to show if the benefits of local instead of global small spaces pays compared to the additional management overhead.

9.3 Hiding the shootdown problem

When a page is unmapped, the translation of that page's virtual address is no longer valid. However since the TLB caches the least recently used translations, the translation might remain in the TLB and therefore has to be invalidated.
The Intel Pentium SMP has a TLB per processor and in the same time, does not support cross-processor invalidation of TLB entries. Since the unmapped page, that probably was mapped to several different address spaces before being unmapped, might have a valid translation in each processor's TLB.
To sum up, we have to invalidate the TLBs of each processor in the system. At least each where the translations might be cached in. Since there are no cross-processor TLB invalidation operations, we have to trigger this operation processor locally i.e. with an Inter Processor Interrupt and invalidate the TLB entries in the handler routine.
The μ-kernel performs the unmap operation in behalf of the pager thread, that initiated the operation. Which means, that this particular thread is accounted for the time needed to perform the unmapping, which is not bound, since once a page is owned by a task, it can be mapped in theory infinitely often into other tasks. In reality, since mapping needs some time, the number of mappings is limited by the interval between mapping the page and unmapping it again which might be arbitrarily long.
When shooting down the TLBs in the system, i.e. invalidating the changed translations, the time can't be accounted to the unmapping thread, since because of *design decision 1* timeslices are processor local. So minimizing the interruptions on the shoot down processors is the target of optimization.

1. When knowing, that a translation is for sure not cached on a specific processor, this processor can be excluded. But detecting where a translation is cached is not such an easy task, because the Pentium's TLB contents can't be read by software.
   As a first approximation, the current address space numbers of each processor can be published and compared to the set of spaces, where the page is going to be invalidated. Since the TLB is flushed on a switch to the next hardware address space, the invalidation triggering IPI can be avoided if the current space is not one of those in the set. However, additional action is required for small spaces, because when switching to a small space, the TLB is not invalidated. So the condition changes to: If the large or one of the small spaces is included in the set of address spaces, where the page has to be unmapped in, the TLB has to be invalidated.

2. Since the TLB is going to be flushed anyway, when switching to the next hardware address space, the shootdown operation can be delayed to that point and skipped. But since multiple switches between small spaces can occur before switching to a large space, the unmap operation can be delayed arbitrarily long.

- ***Implementation decision 5:*** *For the time being, a TLB flushing IPI is sent to all processors on an unmap operation.*

Future work has to evaluate both possibilities.
The following two sections are based on the idea to support real-time applications on symmetric multi processors, i.e. applications that need a guaranteed amount of execution time to fulfill their task correctly. To guarantee these demands, the duration of certain operations like memory accesses need to be known.

9.4 Scheduling Memory accesses

In a uniprocessor system only one processor accesses the memory bus (provided no DMA units are present in the system). So the duration of a memory operation is bound by the bank busy time of the chips plus some bus arbitration overhead. In a multiprocessor environment, including uniprocessor systems with multiple DMA units, the memory bus is shared among several CPUs, which makes the duration of memory accesses dependent from the bus load, too. In other words, the processors in the system may down perform themselves and the others due to concurrent memory accesses.
Work done by Frank Bellosa [3] tries to throttle down the other CPUs to guarantee memory bandwidth to a single processor. More theoretical work by Kevin Elphinstone, Jochen Liedtke and myself [1] describes a way to schedule the memory accesses of the different processors, so to split the available bandwidth among all processors in almost arbitrary amounts. In contrast to Frank's paper, gaps are included in the model, i.e. times where the bus idles, because no processor uses it.
Future work has to prove the correctness of the model described in the paper and to evaluate the practicability of this preliminary approach.

9.5 Scheduling Cache Colors

Caches are used to fasten memory accesses to data that is within the cache. Cache coloring is a technique to pin portions of memory into the cache, so it is not replaced and written back to main memory when other data is read that clashes with the cached lines. By pinning certain memory areas in the cache, the time needed for memory operations performed to that area are now for sure cache hits.
Previous work in this area focused on uniprocessor machines. We are not aware of any work that considered cache coloring on SMPs.

10 Conclusion

This thesis showed how to design a μ-kernel like L4 for SMPs with local L2 caches. For those machines or when assuming the SMP kernel that implements threads that might migrate between nodes not sharing its L2 caches, we showed that local fine grain scheduling decisions have to be made. We presented what effects this have on cross processor IPC and showed in detail how to implement cross processor operations in general and IPC in particular.
The implementation part shows how to implement a L4 μ-kernel on an Intel Pentium SMP machine. In detail we focused on the most complex problem of how to implement IPC. Though there are several problems remaining which are partly identified by this thesis we are promising to construct and implement a fully suited L4 μ-kernel for SMPs. Until now, the API has to be changed only slightly, however experience with other SMP architectures and further optimizations for the Intel Pentium respectively a detailed look at the P4 have to prove the concepts presented in this paper.

A Example Spinlock Implementation

Following an example spinlock implementation for Pentium processors:

```
spinlock(lock)

        lock bts [lock],1
        js spin

        critical_section:
        Do work here

        ret

        spin:
        cmp [lock],0
        jne spin

        lock bts [lock],1
        js spin

        jmp critical_section:
```

First we atomically check and set the lock. If we got it, i.e. the lock was not set before the critical section is save to be entered. If not we spin for the lock, just reading and only if clear we try to lock it again. Through this, only when there is a chance to acquire the lock, i.e. the simple read returns a free lock, the lock is tried to set again. This is done because the MESI protocol of the Pentium processors snoops a write, even when the data of a shared line is not modified. Without, each time a processor spins, i.e. each loop iteration, the line would be transferred to modified state on the processor that performed the spin and invalidated on each other.
Additionally to that the loop is optimized such, that for the normal situation, i.e. when the lock is free and the critical section can be entered immediately, the jump is predicted not to be taken (Pentium predicts forward jumps not to be taken if no history exist).

B Glossary

References

[1] Kevin Elphinstone, Marcus Völp, Jochen Liedtke
Preliminary Thoughts on Memory Bus Scheduling
in the proceedings of the *European SIGOPs Workshop 2000*

[2] *Intel Architecture Developer Manual IA 32 Vol. 1 - 3*
http://www.intel.com

[3] Frank Bellosa
Process Cruise Control
throttling memory access in a soft realtime environment
Poster Session SOSP 1997

[4] *Saw Mill Linux*
http://research.ibm.com/SawMill

[5] *K42*
http://research.ibm.com/K42

Additional papers about L4 can be found at http://i30www.ira.uka.de.