Towards Effective User-Controlled Scheduling for Microkernel-Based Systems

Jan Stoess
University of Karlsruhe
Germany

stoess@ira.uka.de

ABSTRACT

With μ-kernel based systems becoming more and more prevalent, the demand for extensible resource management raises – and with it the demand for flexible thread scheduling. In this paper, we investigate the benefits and costs of a μ-kernel that exports scheduling from the kernel to user level. A key idea of our approach is to involve the user level whenever the μ-kernel encounters a situation that is ambiguous with respect to scheduling, and to permit the kernel to resolve the ambiguity based on user decisions. A further key aspect is that we rely on a generic, protection domain neutral interface between kernel and applications.

For evaluation, we have developed a hierarchical user level scheduling architecture for the L4 μ-kernel, and a virtualization environment running on its top. Our environment supports Linux 2.6.9 guest operating systems on IA-32 processors. Experiments indicate an application overhead between 0 and 10 percent compared to a pure in-kernel scheduler solution, but also demonstrate that our architecture enables effective and accurate user-directed scheduling.

1. INTRODUCTION

With the regained interest in μ-kernel based systems from academia and practitioners there today exist a wide range of application scenarios, where μ-kernels are being deployed successfully. That diversity in fields of application is inevitably reflected by a variety of requirements placed on μ-kernel resource management and scheduling. The following collection of examples – which by no means claims comprehensiveness of coverage – points out some of the demands typically placed on today's μ-kernel schedulers:

- *Virtualization environments*, where μ-kernel-like operating system kernels are employed as hypervisors [26, 3, 10, 4], generally demand a hierarchical scheduling scheme with a hypervisor, guest-kernels, and application schedulers being stacked on top of each other. Preferably, the scheme is orthogonal to the policies actually implemented, and permits virtual components to recursively implement their own strategy.

- *Embedded systems*, another key application of μ-kernel technology [7, 16], demand the scheduler to obey hard or soft real-time constraints in addition to functionalities of desktop or server systems. Embedded systems usually express such constraints in form of soft or hard priorities, or timing conditions of individual applications.

- *Multi-core environments*, an emerging technology becoming prevalent in computing systems, make it essential to provide light-weight, fine-grained, and efficient scheduling abstractions, which allow to dynamically exploit available parallelism as much as possible [8, 18, 25].

- *Power-sensitive computing deployments*, like data centers or mobile computers, reposition the traditional performance orientation of scheduling by adding energy and thermal constraints to the objectives to be adhered to by the scheduler [21].

In summary, a qualified modern μ-kernel scheduler should provide recursive, user-controlled, priority-driven, real-time capable, efficient, generic, and flexible scheduling. Moreover, real application scenarios often turn out to be complex combinations of the aforementioned pure examples: Consider, for instance, a mobile appliance, sometimes plugged into a power connector, sometimes running on batteries, which executes a virtualized legacy operating system together with a set of dedicated μ-kernel threads implementing special services. The scheduler of such a system must fulfill some or all requirements at the same time.

The μ-kernel answer to that requirement is its core design principle: Minimize the kernel part of the operating system, in order to permit modularity, flexibility and tailorability of the rest. In providing the notion of kernel scheduling, a μ-kernel would principally contradict to that principle: Kernel-managed scheduling requires a kernel policy that allocates threads to processors. As long as user level resource management conforms to that kernel policy, kernel scheduling is convenient, and enables development of concurrent programs or overlapping computations. However, as soon as user level management claims freedom in scheduling, the kernel policy bars the way to the flexibility the μ-kernel ultimately strives to provide. User-directed scheduling, in turn, exports the control over resource management to applications, allowing them to develop domain-specific solutions. It also removes policy from the kernel, rendering the system more generic and extensible.

In fact, however, scheduling is widely regarded as too entangled with other operating system concepts – control flow, communication, accounting, interrupt handling, to name a

few – to be easily removed without degradinng efficiency. For that reason, virtually all μ-kernels that are of practical relevance employ a kernel scheduler, with the rationale that a μ-kernel must be efficient to be usable at all. Besides arranging oneself with the kernel scheduler, the traditional approach to enable flexibility and tailorability has therefore been to leverage application-level thread packages.

In this paper, we explore the design of a μ-kernel architecture that strives to export *all* scheduling from the kernel to user level. The key idea of our approach is simple: Involve the user level whenever the μ-kernel encounters an situation that is ambiguous with respect to scheduling. For that purpose, we enhance all kernel operations with an additional interface that allows the kernel to resolve the ambiguity based on the user's decision. While the general idea of exporting kernel scheduling to the user is not new, existing approaches are limited to single protection domains. In contrast, our approach relies on a generic scheme that is neutral to address spaces or other protection mechanisms.

The advantage of user level scheduling is obvious: It permits flexible definition of the systems' scheduling behavior. The potential drawback is evident as well: It increases kernel-user interaction and thus potentially reduces efficiency. Additionally, given the complexity and entanglement of scheduling, any effort to export scheduling to user level is likely to reveal unforeseen problems impairing or undermining its viability. However, as μ-kernels and systems built on their top are becoming more prevalent, the overall demand for flexible resource management raises, and with it the demand for true user-controlled scheduling. We expect our work to be an insightful step towards the development of more generic and flexible μ-kernel scheduling schemes.

Rather than proposing a radical new μ-kernel design – and potentially throw away years of research on and experience with μ-kernel based systems –, we strive to explore if user-based scheduling can be developed in a manner that is compatible to existing μ-kernel designs. In particular, we are interested if a μ-kernel can support user level scheduling while maintaining generic and efficient communication as its most important mechanism. We have therefore based our new architecture on the L4 μ-kernel. While such a decision closely ties some of our design choices to L4, we believe that our findings still apply to μ-kernel architectures in general.

To evaluate our architecture, we have developed an L4-based virtualization environment that supports user level scheduling. The environment runs Linux 2.6.9 guest operating systems on IA-32 processors. Our experiments indicate an application overhead between 0 and 10 percent compared to a solution involving a kernel-scheduler, but on the other hand, demonstrate that our architecture also enables effective and accurate user-directed scheduling.

Since our evaluation presently focuses on a single scenario, the question is still left unanswered whether our approach is generic enough to perform effectively in different application domains. We present an analysis of potential limitations, as they may occur when trying to apply our scheme to other problem domains. We also reason about ways to overcome those limitations.

In the remainder of the paper, we first present our general design principles for user-based scheduling in μ-kernels in Section 2. We then present the application to L4 in Section 3, and a performance evaluation in Section 4. We present insights and limitations in Section 5, and discuss related approaches in Section 6. We finally conclude in Section 7.

2. USER LEVEL SCHEDULING FOR μ-KERNELS

The basic idea of our new architecture is a very simple one: The μ-kernel does not pursue any kernel scheduling anymore. It jettisons all scheduling-related context information such as time-slices, priorities, or run queues, and reduces thread semantics to the notion of execution state. The timer interrupt becomes a "normal" interrupt again, treated like all other external interrupts; other timing semantics vanish from the kernel as well.

Lacking in-kernel scheduling, our new μ-kernel will execute the currently running thread (in its notion of execution context) unconditionally, until a blocking kernel event or operation occurs. A design choice of our scheduling architecture is that it defines operations where a thread blocks on a specific destination thread to be implicit and unconditional processor time transfers; that is, the current thread transfers both control and processor time to the destination. As an example, a thread waiting to communicate with another thread will implicitly donate its processor time to the destination.

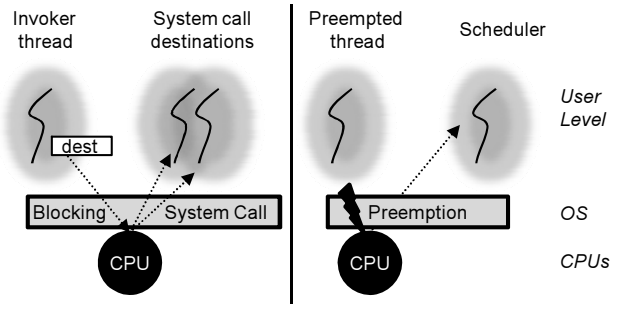

Figure 1: User-Level Scheduling Design. The kernel treats blocking system calls as implicit control transfers; the invoker resolves scheduling ambiguities by selecting the destination thread. Preemptions translate into notifications of a user level scheduler.

Obviously, such a design does not resolve the ambiguity of kernel operations that unblock multiple threads. In case the reason of the operation is an exception or interrupt – an in-kernel event, in other words – the kernel may resolve the ambiguity itself. For all user-initiated operations, however, the kernel must provide an interface that hands over the control over the ambiguous situation to the invoker thread (Figure 1). Such an interface must allow the invoker of a blocking kernel operation to designate, among those threads that are runnable afterwards, the destination thread to be granted the processor. Furthermore, as there can be only one runnable thread at a time, the interface must translate the semantics of a "runnable" thread that is not granted the processor into a notification of a user level scheduler component; the thread itself must be blocked until the scheduler entity has handled that event.

3. APPLICATION TO L4

We have developed a user level scheduling architecture based a recent implementation of the L4 μ-kernel, code-named *L4Ka::Pistachio* (We will hence use the term L4 for both the abstract kernel and our concrete implementation). L4 is a state-of-the-art μ-kernel, which provides a minimal set of abstractions designed to build extensible systems on top [11]. It has three core abstractions: threads, address-spaces, and inter-process communication (IPC). In the following section, we first describe the scheduling-relevant aspects of the original L4 version. We afterwards present the changes we have made in order to provide effective user level scheduling. In particular, we elaborate on the mutual implications of scheduling and IPC.

3.1 Original L4 Behavior

In the following, we describe the original functional behavior of L4 with respect to threading, communication, and scheduling, as they are all important to understand the nature of scheduling and its implications to μ-kernel operations. We then analyze the original L4 model with respect to its deficiencies to provide full user level scheduling.

3.1.1 Threading

In the original L4 implementation, threads provide the context for three different operating system concepts: execution, communication, and scheduling. In their first role, they serve as the basic abstraction of user level control flow, by referring to execution state such as instruction and stack pointer, or general purpose registers. In their second role, threads serve as endpoints for the kernel IPC primitive. L4 therefore associates IPC state with each thread, which keeps track of attempted or ongoing IPC operations and their particular form and content. In their last role, L4 associates information such as time slice lengths, time quanta, or priorities with each thread, and employs an internal scheduler that dispatches the threads using a default policy.

3.1.2 Communication

L4 IPC is a rendezvous-based, synchronous communication mechanism. L4 offers both a send and a receive operation; while L4 permits sending to unique destinations only, it allows the use of wild-cards as receive destination. As system calls are expensive, and for reasons of atomicity, L4 also offers coalesced send and receive operations. We can distinguish four types of IPC operations: i) `send(to)`, a single send operation to a specified destination, ii) `receive(from)`, a single receive operation from a specific or an arbitrary thread, iii) `call(dest)`, a coalesced send and receive to the same destination, iv) `send_and_receive(to, from)`, a coalesced send and receive operation with different destinations. If the receive destination is a wild-card, the operation is dubbed `reply_and_wait(to)`.

L4 supports message contents of different complexity, with registers, strings, and virtual memory mappings as transferable objects. Finally, L4 supports the specification of timeouts for blocking IPCs.

3.1.3 Scheduling

L4's kernel scheduler resembles traditional process schedulers of monolithic operating systems. All runnable threads are enqueued into a processor-local ready queue. Timer interrupts, blocking kernel operations, and user-initiated thread switches trigger the invocation of the L4 kernel scheduler, which chooses the next running thread based on the kernel policy. For reasons of efficiency, L4 does not invoke the scheduler during IPC calls; instead, it employs a time donation model, where the blocking thread passes its remaining time slice to the partner [16].

L4 provides an interface that allows user level programs to adjust scheduling parameters. For that purpose, L4 arranges threads into a hierarchy, by associating a scheduler thread with each thread. In the current implementation of L4, the hierarchy is not required to be strict; for instance, a thread may act as its own scheduler. L4 permits schedulers to freely set the parameters of their subordinate threads, with the restriction that priorities may not exceed the own priority of the scheduler.

3.1.4 Analysis

While user level schedulers can tweak the kernel scheduling policy, core scheduling behavior such as the policy itself cannot be changed. As a result, a scheduler that requires a policy different to the default kernel policy needs to pursue complicated steps to implement it. The root cause is that there is no visible difference between the thread currently running and other runnable threads. For a user level scheduler, all its subordinate threads appear to be running at the same time. To implement its own strategy, a scheduler must thus prevent L4 from selecting the "wrong" thread, by making sure that all threads except the one it itself designates to run are not runnable. Assuring that is awkward and inefficient; threads may change their own state when invoking kernel operations, and the scheduler requires a complex state machine to ensure correct states while on-going operations are handled gracefully.

L4's hierarchical scheduling design even fortifies that deficiency: Since L4 does not allow a high-level scheduler to control whole scheduling subtrees, and since threads may reside in various states depending on their own behavior and on external events, high-level schedulers need to either trust their successors that they have taken care of well behaving thread scheduling, or they need to ensure the behavior itself, per thread. Both cases are unsatisfying: The former is unsafe and contradicts the trust chain, while the latter burdens all levels of scheduling with the same redundant complexity.

To summarize, the effects and decisions of the kernel scheduler are intransparent and opaque for user level schedulers, despite their nominal status as schedulers.

3.2 Preemption Messages – User Level Scheduling for L4

To overcome the scheduling deficiencies of L4 we have developed an L4 version that supports user-based scheduling. Our new version uses threads as a container for execution and communication, stripping away the notion of scheduling. There is no scheduling state or policy in the kernel anymore; instead, L4 runs the current thread unconditionally until the next event occurs that causes its preemption. To give user level schedulers full control over the dispatch-

ing, L4 vectors out the preemption of a thread to its associated scheduler thread, by means of a preemption IPC (Note, that the notion of preemption IPC already exists in the original L4 kernel, but with different semantics related to time-quanta [23]). A preemption IPC is a coalesced send and receive operation to the scheduler, where the preempted sender automatically blocks waiting for a reply message. To re-grant processor time, the scheduler responds by sending back an IPC reply message, which again results in the destination thread running until the next preemption occurs (see Figure 2).

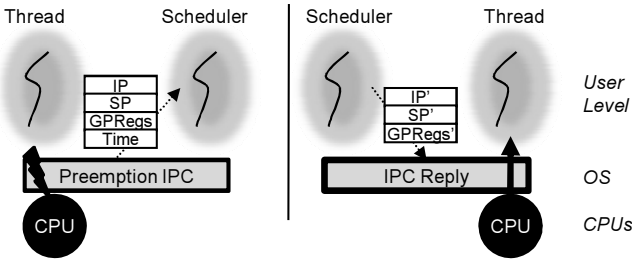

Figure 2: Preemption IPC and reply

For reasons of convenience, a preemption IPC contains the thread's current execution state such as instruction and stack pointer and general purpose registers. A reply may contain updated state, which is then transparently installed into the thread's user frame. Transferring the whole execution state allows the schedulers to inspect thread execution context; updating the state with the reply eases typical scheduler operations such as sending signals or switching to a different user level thread.

A principle of user-controlled scheduling is that there is only one thread running at a time per processor. This is reflected in L4 in that all other threads are blocked in a kernel operation waiting for a time donation. The current thread is allowed to run until it deliberately blocks, or until it is forcedly preempted by an external event. Forced preemptions occur whenever a thread would stay runnable but is preempted without its own agreement. L4 then synthesizes a preemption message on behalf of the thread sent to its scheduler. Deliberate preemptions occur whenever the thread invokes a blocking kernel operation. If the blocking operation has a unique destination, L4 will treat the call as processor time donation, where the current thread transfers its processor time together with the call. As the invoking thread is aware of its own blocking, L4 does not send any preemption IPC. The typical such operation is a blocking IPC call, and threads are expected to use IPC to donate processor time among each other.

To resolve the ambiguity that arises when a kernel operation unblock multiple threads, the kernel allows the thread that has invoked the operation to designate the destination thread to be granted the processor. We therefore enhance such kernel operations with a set of bits that permit the invoker to specify the eventual processor owner.

3.3 Preemption and IPC

In the original L4 version, IPC involves threads in all their roles: as execution, as communication contexts, but also as scheduling contexts. Specifically, the transfer of an IPC requires a scheduling decision to be made whenever it unblocks a thread with a higher importance than the ones running, or when it unblocks more threads than it blocks. The original version of IPC either invokes the kernel-level scheduler when a decision is required, or resorts to a simpler shortcut policy that circumvents the potentially complex kernel scheduler.

We instead treat IPC according to our explicit model: All cases that block the current thread and activate one destination become an implicit time donation. All other cases are resolved based on the invoking thread's decisions. The implications of user level scheduling on IPC thus depend on its type and are as follows:

blocking send or blocking receive In case of blocking `send` or `receive` operations, the invoker immediately switches to the idle thread, since no other thread is running. Note, that our approach also changes the semantics of the idle thread; see Section 3.4 for details.

send and blocking receive Here, the sender successfully sends a message but is then blocked waiting for the receive destination. The invoking thread grants the processor to the destination.

non-blocking send or non-blocking receive After successful single `send` or `receive` operations, the invoker unblocks the destination thread while staying runnable itself; the invoker decides if it wants to grant or retain the processor; the losing thread sends a preemption IPC.

non-blocking send and receive The successful coalesced `send and receive` variants cause a situation where both send and receive partner are runnable and the invoker is waiting for the receive destination. The invoker again designates the processor grantee among the two IPC destinations, the other sends a preemption IPC.

3.3.1 Sender-based Message Transfer

For reasons of implementation, the kernel performs IPC message transfers always from the sender to the receiver thread. Since L4 also offers to coalesce operations, IPCs may become nested into each other. A side-effect of the nesting is that a thread, being unblocked by its send partner in order to transfer a message, may immediately afterwards try to receive from – and switch to – a completely unrelated thread (Figure 3).

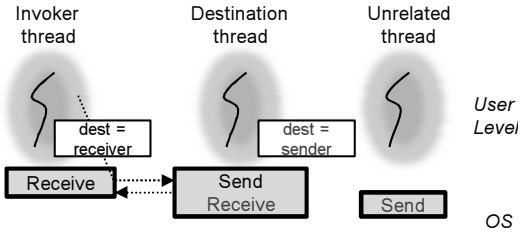

Figure 3: Nested IPC chains. To avoid activating unrelated threads, the kernel always switches back to the invoker of the IPC.

With respect to user-directed scheduling, such a behavior is undesirable, since it does not permit receiving threads to retain the processor after the message was transferred. We solve this problem by switching back from the sender to the receiver of the first IPC rendezvous in those cases, regardless of the destination specified at the time of invocation. L4 therefore checks whether the send operation blocked, in which case the sender switches back to the receiver.

3.3.2 Recursive Preemption

So far, we have discussed the implications of user-controlled scheduling to IPC. However, as our architecture leverages IPC to vector preemptions, IPC semantics affect user level scheduling as well. The most important aspect of this duality is that a preemption IPC activates the waiting scheduler, and leaves the kernel in a situation where two threads become runnable, the thread having caused the preemption (hence the *preempter*) and the scheduler.

The most intuitive solution to decide this ambiguity is to give the preempter preference over the scheduler, since it was the preempter that caused the situation in the first place. However, as a result, the scheduler itself will generate a preemption message to a higher-level scheduler, which will in turn notify its scheduler, and so on, until either a scheduler is not waiting for the message, or the top-level scheduler in the hierarchy has been reached. The root-level scheduler, being non-preemptable by definition, eventually preempts the original preempter, causing a chain of preemptions again.

Such recursive preemption may, at a glance, seem ineffective. It is, in fact just a consequence of the hierarchical scheduler arrangement in L4. Shortcutting any preemption IPC implies that the kernel is left in a state where more than one thread is runnable at the same time, and eventually needs to decide which of the threads are dispatched. To permit scheduling to be user-controlled, our approach prevents this situation on principle.

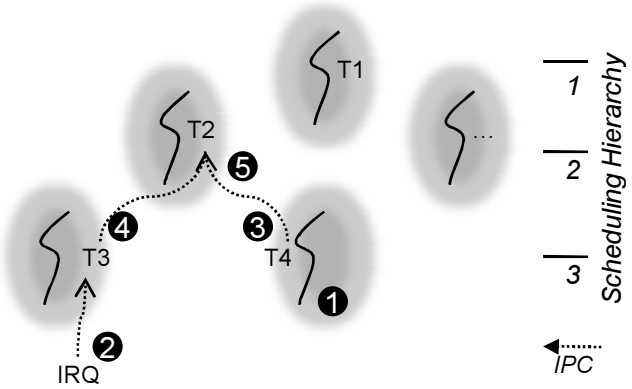

Figure 4: Recursive Preemption Message Chain. While executing (1), T4 is preempted by an interrupt delivered to T3 (2). It sends a preemption IPC to its scheduler T2 (3); since T2 is a scheduler of T3 as well, T3 also sends a preemption IPC to T2 (4), and T2 starts executing (5).

Our new architecture implements hierarchical scheduling, including recursive sending of preemption messages; however, we still found room for an important improvement. Giving the preempter preference will result in the preemption logic always ascending to the root scheduler, while in principle, ascending to the lowest common scheduler of the preempter and the destination thread would be sufficient. We can shorten the preemption chain by checking, during preemption, the hierarchical scheduling relationship between the preempter and the destination scheduler. By keeping track of each thread's scheduling hierarchy level, we can cheaply determine relationships between the preempter and the scheduler in most cases, without traversing the scheduling hierarchy. By default, the logic will give preference to the preempter; however, in case the scheduler thread turns out to be a scheduler of the *preempter* as well, the logic will favor the scheduler (see Figure 4). In that case the preempter itself initiates a preemption IPC chain, until the aforementioned scheduler is reached. Having received a message from the original thread in the first place, the scheduler cannot receive the second other message immediately. Nevertheless, it is also the only thread running, and the whole hierarchy is in a consistent state again. Note, that the scheduler needs to take into account that there are potential pending preemptions after having received a preemption message. It can cater for that situation by receiving potential message with a zero timeout IPC.

3.3.3 Timeouts and Wakeup Queues

L4 originally introduced timeout semantics for IPC, for different reasons: Timeouts prevent a thread from being blocked infinitely by an untrusted IPC partner [11, 13]. Timeouts further serve as means to implement error recovery; finally, they simply keep a thread blocked waiting until a certain period of time has elapsed. However, the benefits of timeouts with respect to safety has been questioned recently [19]; some current L4 designs even refrain from providing timeout logic at all [5].

Our new architecture also removes any timeout logic from the kernel, allowing the user only to specify either `never` and `zero` timeouts only. As a consequence, IPC is guaranteed to either block infinitely or to return immediately. Our rationale to remove timeouts is obvious: Export all timing-related handling to user level. Maintaining timeouts implies that threads are potentially unblocked; and removing timeouts relieves the kernel from having to decide the next thread at such events. Moreover, kernel-based timeouts are conceptually superfluous, since they can be implemented at user level, via a third party that cancels operations after the desired period of time has elapsed. At present, our approach does not implement such a protocol, however; we leave the question and potential performance problems as an area of future work.

3.3.4 IPC Efficiency

IPC efficiency is widely regarded as the deciding factor for a μ-kernel based system, and much effort of research on L4 has been dedicated to ensuring or improving its performance [12, 14, 11]. It was one of the goals of this project to investigate if developing effective user level scheduling was possible without having to throw away all the findings of research on efficient communication. The following paragraph is devoted to a short discussion on that matter.

Since we treat blocking kernel operations that activate a destination thread as implicit time donations, the consequences of user-controlled scheduling on IPC are limited: All blocking IPC types with just one destination are unaffected and retain their performance[1].

Fortunately, the `call()` operation, which is probably used most frequently in a RPC-based system, falls under the category of implicit time donations. Its counterpart, the `reply_and_wait()` operation, which sends a message to a specific thread and then waits for incoming messages, falls only partly under that category: In case of no pending send request, the invoker immediately switches to the send destination without additional preemption logic. In case requests are pending, the invoker needs to decide between the send and the prospective receive partner, which implies that one thread is preempted. Finally, the `send_to()` and `receive_from()` operations requires the invoker to decide if it wants to retain or grant the processor away its processor, again with the result of preemption messages being generated.

We consider `send_to()` and `receive_from()` as unproblematic; `receive_from()` only leads to problems if the partner does not block itself, which is the unlikely case in a RPC-based system; `send_to()`, in turn, can easily be replaced by appropriate substitutions: Sending and retaining the processor with a shared user variable, and sending and granting the processor with a `send_and_receive()` operation that blocks waiting for a fresh time donation from the scheduler.

The `reply_and_wait()` operation, in contrast, implies inevitable overhead on loaded server systems with a backlog of pending requests, since the user level scheduling logic generates preemption messages whenever two clients become ready. Besides avoiding the backlogs via appropriate user level design, we see two different ways out of that dilemma. The first alternative is to construct a μ-kernel solely based on procedure call semantics (as realized, e.g., in K42 [2]). A second alternative would be to retain a recipient-based communication scheme but to give user threads hints on the state of their of wait queues. Our previous research indicates that a simple mechanism such as event logging may be sufficient to export such kernel data efficiently [22]. That way, a server that has finished a request can inspect the state of its send-queues before issuing a `send_and_receive()` operation. In case of a pending requests, it can either first receive that request (via a single `receive_from()`, which does not incur preemption, provided the client uses a blocking `call()`), or skip the pending request, send the reply to the client, and then wait for a time donation from its scheduler using `send_and_receive()`. We have not implemented a solution yet, but are currently investigating the latter approach.

3.4 Other Kernel Operations

User level scheduling does not only change the semantics of IPC, it also affects all other kernel operations that may cause threads to become unblocked. Fortunately the number of such operations is very limited; effectively, there are only three such operations in L4 besides IPC: The switching to an idle thread, and the `ThreadSwitch` and `ExchangeRegisters` system calls. We will discuss them in the following section. For the sake of completeness and importance, we also present the new semantics of the exception and interrupt handling, although all changes to their handling are a direct consequence from aforementioned changes in the IPC semantics.

3.4.1 Switch to Idle

As most other kernels, L4 features an in-kernel, per-processor idle thread that is invoked when no other thread is runnable, and runs until the next external interrupt occurs. In contrast, our modified L4 version does not contain such a special idle thread anymore. Rather, it exports the idle transition by means of a special IPC message sent to the root-level scheduler of the respective processor. Sending such notification is not strictly necessary, since our model allows current thread to do whatever it wants to do with its processor time, including "nothing". We see our idle IPC message protocol mainly as a convenient method that allows top-level schedulers to determine idle processors. To allow switching to low power modes, L4 additionally permits the top-level scheduler to execute special idle instructions such as `hlt` on IA-32 based processors.

3.4.2 Thread Switch

L4 originally offers a `ThreadSwitch` system call that donates the current time slice from the caller to the callee; if no destination is specified, the in-kernel scheduler selects the next thread to run. For our new architecture, `ThreadSwitch` is a bogus operation, since only the current thread is runnable; for reasons of compatibility `ThreadSwitch` is still available, but now sends a preemption IPC to the caller's scheduler.

3.4.3 Exchange Registers

The `ExchangeRegisters` system call allows a thread to read or modify parts of the execution and communication state of another thread, provided both threads are executing within the same address space. To that end, it also allows the invoker to suspend or resume other threads. Our new kernel therefore allows the invoker of `ExchangeRegisters` to decide if it wants to retain the processor or not. The kernel synthesizes appropriate preemption messages on behalf of the respective other thread.

3.4.4 Interrupts and Exceptions

L4 generally handles interrupts and exceptions by means of IPC. For each external interrupt line, L4 creates an in-kernel interrupt thread, which will send a special IRQ message to an attached handler thread whenever the interrupt occurs. Similarly, whenever a synchronous processor exception occurs, L4 synthesizes an exception message on behalf of the faulting user level thread to a designated per-thread exception handler.

The implications of user level scheduling on interrupts all arise from the novel semantics of hierarchical scheduling dealt with in the IPC path. Whenever a thread is preempted by an external interrupt, L4 sends a preemption message to its designated scheduler in addition to the original interrupt message sent to the handler. Interrupts therefore trigger to

[1] A side-effect of that model is that it leaves the scheduler unnotified of deliberate blockings; it is up to – and in the interest of – the user threads to agree on a protocol with the scheduler to inform it about such preemptions, if necessary.

a chain of preemption messages according to the protocol described in Section 3.3.2. In contrast to interrupts, exceptions are synchronous, and L4 delivers a blocking IPC to the associated handler, with no additional preemption logic being involved.

4. EVALUATION

To evaluate our novel user level scheduling model, we have developed a virtualization environment with support for true user level scheduling semantics, based on the existing afterburner project. The afterburner is an approach to provide virtualization on top of L4 [10]. Our new environment currently supports a modified version of Linux 2.6.9 on IA-32, with support for both physical and virtual multi-processing.

Since user level scheduling incurs additional IPCs and potentially additional protection domain switches, we considered the efficiency of our architecture as the most interesting evaluation criteria. In the following section, we try to shed some light on the performance of our new architecture. We first describe the basic structure of our virtualization architecture, as it is relevant to user level scheduling. We then present a set of micro- and macro-benchmarks intended to show the overhead of effective user level scheduling in an IPC-based μ-kernel.

4.1 Virtualization Architecture

The afterburner approach can be described as an interface layer that translates sensitive guest operating system instructions into API invocations of the underlying virtualization architecture. The afterburner environment consists of the L4 μ-kernel as the hypervisor, and user level components that implement the virtualization services. For performance reasons, a large fraction of the user level code executes in-place, within the address-space of guest operating systems. If necessary, for instance for reasons of security, the in-place part calls into an external module named resource monitor, which runs in a separate address space and has extended privileges.

4.2 Virtual Machine Scheduling

With respect to scheduling and threading, the original afterburner implementation differs substantially compared to our new version. The original afterburner spawns an L4 thread per guest user thread. The threads are then scheduled intransparently, according to the internal round-robin strategy of L4, and without the guest operating system taking notice.

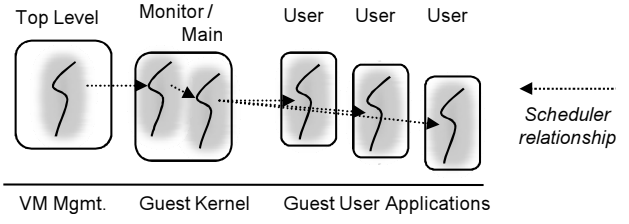

Figure 5: 3-Tier Virtualization architecture

Our new architecture, in contrast, uses a completely redesigned threading model, which enables scheduling that is fully directed by guest operating system kernel. The architecture constitutes a three-level scheduling hierarchy (see Figure 5). At the highest level, a per-processor top-level scheduler thread runs in the address space of the privileged resource monitor and serves as scheduler for virtual processors. At the second level, each virtual processor is represented by two threads representing the guest kernel. One guest kernel thread, named *main*, serves as the execution context for the virtualized guest operating system code; the other thread, named *monitor*, acts as the in-place resource manager and scheduler of main. Finally, at the third and lowest level, the afterburner spawns a thread per user level address space and virtual processor, to execute guest user code. Each user thread on a virtual processor is scheduled by the main thread on that processor.

Our new scheduling model dictates that all user threads except the one currently running are waiting for their main thread, either for a preemption reply, or for a reply to an exception or page fault. The virtualization logic translates the `iret` instruction, issued by the guest operating system on the main thread to transfer control to a user level program, into an IPC reply to the waiting user level thread representing that program. Preemptions of the main thread, in turn, are serviced by the monitor thread. The monitor itself is finally scheduled by the privileged top-level scheduler, which associates itself with all interrupts and dispatches them appropriately to the guest operating systems.

To illustrate the procedure of hierarchical scheduling, let us consider the handling of the timer interrupt: At each timer tick, the current user program thread is preempted by the L4 kernel thread representing the hardware timer interrupt. Thus it sends a preemption message to its corresponding main thread. According to the hierarchical protocol, the activated main thread is preempted itself by the interrupt thread, and sends an IPC to the monitor thread. The monitor finally sends a preemption message to the top-level scheduler, where the chain ends. The activated top-level scheduler receives both the interrupt message from L4 and the preemption message from the monitor thread, before it selects the next virtual processor to be granted the processor. It then dispatches the next virtual processor by sending an IPC reply to the corresponding monitor thread, which was blocked waiting in the same manner as the monitor thread preempted now.

4.3 Performance

We deemed several questions as relevant with respect to performance: How expensive is preemption in terms of the basic absolute overhead? How often does preemption happen in a realistic scenario? Are the costs of the preemption logic reflected in overall application performance? In the following, we present measurements to answer those questions. We conducted all measurements on a 3 GHz Pentium D830 with 2 GByte memory and an attached Intel E1000 Gigabit network interface. Since the original afterburner version does not support multi-processing, we used only one of the two cores to ensure fair comparison.

To quantify the basic overhead of adding user level scheduling logic to the IPC path, we compared the `call()`-type IPC of the original version against the same operation on our new

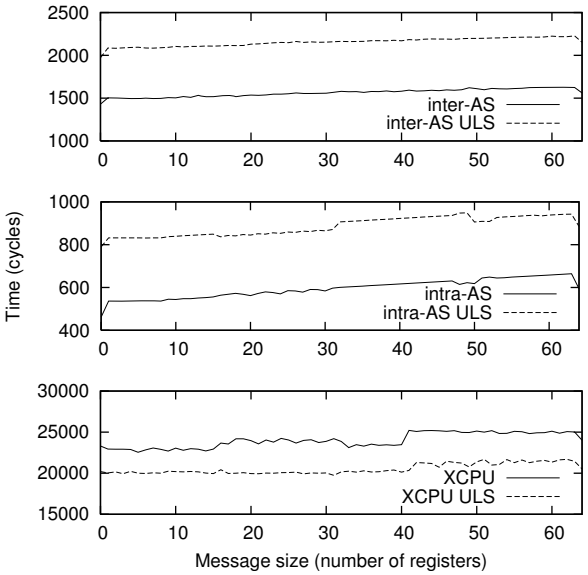

Figure 6: IPC costs of L4 with support for user level scheduling compared to the original version.

L4 version. We leveraged a ping-pong IPC micro-benchmark that creates a pair of communicating threads that perform message transfers back and forth. The benchmark measures round-trip time for different message sizes; since transfer times are short, the benchmark conveys messages repeatedly. Figure 6 lists the results in cycles, for inter- and intra-address space, and for cross-processor IPC. The results state the costs of additional scheduling logic to be at most about 50 percent of the native IPC costs. Since our design does not change the `call()`-type IPC semantically, we attribute the performance degradation of the single-processor IPC to the increased complexity of the IPC code path and to under-optimized code. We currently have no explanation for the performance improvement of the cross-processor IPC. Altogether, we consider the results to be not negligible, but still tolerable.

In the next experiment, we determined frequency and direct costs of hierarchical preemption in our three-tier scheduling environment, as well as the overhead of the preemptions on some selected benchmark applications. To measure direct preemption costs, we measured the latency between the activation of the L4 kernel interrupt handler and the actual execution of the user level root scheduler thread associated with the handler.

Since our environment does not trigger preemptions during synchronous execution, the preemption frequency solely depends of the number of interrupts and the levels in scheduling hierarchy. To determine realistic values for the frequency, we generated I/O load by executing the `netperf` benchmark on a uniprocessor Linux guest operating system installation, from the Gigabit NIC to an external client. We also generated processing load, using two different applications. Firstly, we ran `cpu`, a small user program that loops forever counting bits of a dummy value, which results in guest application code being executed permanently. Secondly, we ran the Apache `ab` server benchmarking tool with a concurrency level of 128, and with the 16 clients and the server residing within the guest, resulting in substantial processor load to handle the traffic on the loop-back device.

Workload	F	L	Workload	F	L
IDLE	0	1.15	NET+CPU	48	3.26
CPU	3	2.78	AB	2	4.26
NET	16	2.22	NET+AB	38	3.39

Figure 7: Preemption frequency F [$\frac{\#}{tick}$] and interrupt latency L [μs], under different workloads.

Figure 7 lists the results per timer tick (currently 2 ms). In case of an idle processor, no preemption occurs; the interrupt latency reflects the basic costs of delivery. In the case of `cpu` running, the number of preemptions per tick is equal to the number of hierarchies up to the guest user applications (i.e., 3); the interrupt latency raises accordingly. With `netperf` running, the number of interrupts raises, but given `netperf` is mostly I/O-bound and no other application are running, interrupts often occur when the processor is idle; thus the number of preemptions is equal to the number of interrupts, but the interrupt latency drops. Accordingly, in the next case where `netperf` and `cpu` execute concurrently, the preemption frequency is three times as high, and the interrupt latency raises again. In the last two cases `ab` causes execution of a large amount of guest kernel; the guest is therefore often preempted at level 2 in the scheduling hierarchy. We attribute the higher interrupt latency to the increased workload size of `ab` compared to `cpu`.

Figure 8 shows the throughput of `netperf` and the request rate of `ab` compared to the original infrastructure and to a native Linux installation. The throuhgput of `netperf` is roughly comparable to the original architecture, while the rate of `ab` drops by about 10 percent to 28 request per ms.

	ULS		Original		Native	
Workload	T	R	T	R	T	R
NET+CPU	933	-	930	-	940	-
AB	-	27.8	-	31.3	-	146.2
NET+AB	934	13.4	151	29.6	940	97.0

Figure 8: NetPerf throughput T [$\frac{MB}{sec}$] and Apache request rate R [$\frac{\#req}{ms}$] of our architecture, compared to the original infrastructure and to native Linux.

The benefits of user level scheduling are most evident when `netperf` and `ab` run concurrently: With our new architecture, `ab` throughput drops by 50 percent, while `netperf` stays undisturbed. This is conforming to the outcome of the native installation, where `ab` throughput drops while `netperf` performs alike as well. However, in the original version, `ab` throughput stays equal, while `netperf` throughput drops dramatically by 84 percent! We can clearly attribute that behavior to the L4 kernel scheduler counteracting the intentions of the Linux guest scheduler.

Altogether, the results lead us to the following conclusions: hierarchical preemption incurs between 1.4μs and 2.1μs overhead per level; in a realistic three-tier hierarchy the average overhead amounts to about 156 μs per timer tick of 2 ms

maximum overhead under high I/O load, or 7.8 percent. At application level, our architecture performs between 0 and 10 percent worse to the original in-kernel solution, but, on the other hand, permits true and accurate user-directed scheduling.

5. INSIGHTS AND LIMITATIONS

So far, we have evaluated our user level scheduling approach solely for a virtual machine scenario. We have demonstrated that the approach permits flexible, and completely user-defined scheduling, provided the user application and communication model is similar to the one presented. However, the question whether our approach performs equally effective in a different scenario is still left unanswered. Given the wide range and technical complexity of application scenarios and of a modern μ-kernel, it was impossible for use to substantiate that our approach is generic and flexible enough to fit every imaginable requirement. In the following section, we therefore stick to a theoretical analysis of potential limitations, as they may occur when trying to apply our scheme to other problem domains. We also reason about ways to overcome those limitations.

One of the key design decisions of our approach is that it leaves the decision how to use processor time completely up to the thread currently executing, rather than to its scheduler. Furthermore, the approach strives to render all kernel primitives scheduling neutral in that it allows the sender to specify the grantee of the processor at its own will. The reasoning behind that decision is that the current thread owns the processor until the next preemption; it could even use its time for performing useless computation during that time. However, our model implies that, for global scheduling policies to work, each thread must implement parts of the scheduling policy when activating other threads via system calls. Also, in case there is no control over the execution of system calls, any scheduling policy may easily be undermined. Consider, as an example, a scheduling policy with fixed priorities; with our scheme, nothing prevents a high priority thread to activate a low priority thread via IPC, even in presence of preempted medium priority threads waiting for processor time.

From a resource management point of view, our user level solution is not very different to an in-kernel solution, since latter needs to track and schedule the correct thread after system call executions as well. In our eyes, there is no general problem in shifting that task to a scheduling library running at user level – apart from potential performance implications, which we already have discussed. Still, our system presently lacks appropriate control over arbitrary control transfers and their implications on priority inversion and time isolation. Priority-based real-time scheduling schemes hence remain an open problem not addressed by our current approach.

A further implication arises from the recursive scheme dictating the order of preemption messages generated when a hardware preemption occurs. That scheme may be, to some extent, considered as kernel scheduling policy – something that we originally strived to remove from the kernel. Similarly, the decision to treat blockings on destination threads as implicit processor grants may be viewed as a default kernel policy. However, we see those design decisions uncritical: The special semantics of blockings on single threads solely stem from performance considerations; we see no design limit that hinders application of our scheduling interface extension also for those types of calls. Our recursive scheduling, in turn, merely reflects the hierarchical scheduler arrangement intrinsic to L4. The scheme, however, is still generic in that it allows user level components to arrange the scheduling hierarchy according to their demands. To give an example, a global in-kernel scheduler could be simulated at user level, using a flat hierarchy with one global scheduler per processor.

6. RELATED WORK

There exists a considerable body of research on the problem of directing kernel scheduling from user level [2, 24, 1, 15, 9, 6]. To our best knowledge, none of the approaches has proposed and evaluated a μ-kernel-based scheduling approach that relies on a single communication primitive neutral to protection domains for propagating scheduling information.

Two-level scheduling schemes are perhaps the most known effort to user level scheduling. They employ both a kernel- and a user level scheduler, and mechanisms allowing both parties to propagate scheduling events and directions back and forth [1, 15]. Two-level approaches typically require the user level scheduler to reside within protection domain of its threads; as a result, they must employ a residual kernel scheduler and policy that "schedules the schedulers". In contrast, our solution provides a solution that is protection domain neutral and does not require a kernel-level scheduler.

CPU inheritance scheduling very much resembles our approach in that it strives to overcome the rigidity of entangled kernel scheduling in existing operating systems [6]. Like our approach, CPU inheritance scheduling introduces a hierarchical model, where threads wait for timing events and schedule each other using a time donation primitive. CPU inheritance scheduling differs to approach in two regards. First, it employs separate kernel primitives to wait for and donate processor time. We instead leverage a generic communication mechanism, which simplifies the optimization and allows to piggyback scheduling directions onto other kernel operations. Second, CPU inheritance scheduling only presents a preliminary evaluation; we have demonstrated and evaluated our approach based on an existing and realistic scenario.

Finally, there exist other approaches that directly address the problem of scheduling on top of the L4 μ-kernel. The reflective scheduling effort in [17] provides fine-grained real-time scheduling on top of the original L4 kernel scheduler. The Credo approach in [20] proposes a capacity-reserve donation model and extension to the L4 μ-kernel to permit proper resource accounting and real-time properties .

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a novel μ-kernel architecture that strives to export the task of scheduling from the kernel to user level. The key idea of the architecture is to involve the user level whenever the μ-kernel encounters an situation that is ambiguous with respect to scheduling, and to allow the kernel to resolve the ambiguity based on user

decisions. A further key aspect is that our architecture relies on a generic, protection-domain neutral mechanism to vector scheduling state from the kernel to user level.

To evaluate our novel architecture, we have developed a hierarchical user level scheduling architecture based on the L4 μ-kernel, and a virtualization environment running on its top. Benchmarks indicate an application overhead between up to 10 percent compared to a pure in-kernel solution, but on the other hand, demonstrate that our architecture enables effective and accurate user-directed scheduling.

While the question is still left unanswered whether our approach is generic enough to perform effective in different application domains, we still our work as an insightful step towards the development of more generic and flexible μ-kernel scheduling schemes.

Acknowledgements

We would like to thank our shepherd, Michael Hohmuth, and the anonymous reviewers for their in-depth comments and helpful suggestions.

8. REFERENCES

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th Symposium on Operating System Principles*. ACM Press, Oct. 1991.

[2] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. Wisniewski, and J. Xenidis. Scheduling in K42. White Paper, Aug. 2002.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, Bolton Landing, NY, Oct. 2003.

[4] S. Biemueller and U. Dannowski. L4-based real virtual machines - an api proposal. In *Proceedings of the First International Workshop on MicroKernels for Embedded Systems*, Sydney, Australia, Jan. 2007.

[5] Embedded Real-Time and Operating Systems Group. *NICTA L4-Embedded Kernel Reference Manual (Version N1)*. Natioal ICT Australia, 2005.

[6] B. Ford and S. R. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Berkeley, CA, Oct. 1996.

[7] G. Heiser. Secure embedded systems need microkernels. *;login: the USENIX Association newsletter*, 30(6), Dec. 2005.

[8] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st EuroSys conference*, Leuven, Belgium, Apr. 2006.

[9] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7), Sept. 1996.

[10] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[11] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[12] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1997.

[13] J. Liedtke, N. Islam, and T. Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1997.

[14] J. Liedtke and H. Wenske. Lazy process switching. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems*, Schloß Elmau, Oberbayern, Germany, May 2001.

[15] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th Symposium on Operating System Principles*. ACM Press, Oct. 1991.

[16] S. Ruocco. Real-time programming and L4 microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[17] S. Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.

[18] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP enviroment. In *Proceedings of the 2nd EuroSys conference*, Lisbon, Portugal, Mar. 2007.

[19] J. S. Shapiro. Vulnerabilities in synchronous IPC designs. In *IEEE Symposium on Security and Privacy*, Los Alamitos, CA, May 2003.

[20] U. Steinberg, J. Wolter, and H. Härtig. Fast component interaction for real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[21] J. Stoess, C. Lang, and F. Bellosa. Energy management for hypervisor-based virtual machines. In *Proceedings of the USENIX 2007 Annual Technical Conference*, Santa Clara, CA, June 2007.

[22] J. Stoess and V. Uhlig. Flexible, low-overhead event logging to support resource scheduling. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Systems*, volume 2, Minneapolis, MN, July 2006.

[23] The L4Ka Team. *L4 Kernel Reference Manual (Version X.2)*. Universität Karlsruhe, 2004.

[24] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th Symposium on Operating System Principles*. ACM Press, Dec. 1989.

[25] V. Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, May 2005.

[26] VMware Inc. *ESX Server Data Sheet*, 2006.