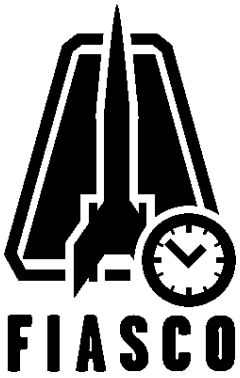
Quality-Assuring Scheduling in the Fiasco Microkernel



Udo Steinberg
us15@os.inf.tu-dresden.de

Technische Universität Dresden
Institute for System Architecture
Operating Systems Group

March 25, 2004

Acknowledgements

First, I would like to thank Prof. Hermann Härtig for the opportunity to work on this project in his operating systems research group at TU Dresden. His lectures sparked my interest in the design and development of microkernels. My special thanks goes to Jean Wolter, with whom I had numerous interesting discussions about the implementation details of the quality-assuring scheduling model. Many ideas that he and I developed together have been incorporated into this work. I would also like to thank my supervisor, Dr. Michael Hohmuth, for providing me with many insights about the internal concepts of his Fiasco microkernel and for his valuable suggestions on how to integrate my ideas into the kernel. Many members of the operating systems group have given me useful advice and helped me test the new scheduler. Adam Lackorzynski allowed me to iron out many scheduler bugs early on by successfully triggering them in best time. Frank Mehnert implemented the low-level driver for the APIC; without his work the one-shot timer interrupt implementation would not exist. Finally, I thank Alexandra Walford, Michael Peter and Marcus Völp for proofreading this thesis. Their valuable comments helped me improve the style of this document tremendously.

Contents

List of Figures

Chapter 1

Introduction

For the construction of a real-time system the underlying kernel must provide flexible mechanisms to implement job scheduling and the corresponding scheduling policies on top of it. The research in the operating systems group at TU Dresden focuses on microkernel-based real-time systems that use the common L4 microkernel interface. Our Dresden Real-Time Operating System [4] is based on the Fiasco microkernel [5], which is a fast and fully preemptible second-generation microkernel written in C++. Fiasco implements protected address spaces, message-based synchronous IPC between threads and a scheduler with multiple fixed-priority levels, whereby the kernel schedules multiple threads on the same priority level according to the round-robin algorithm.

Because priority-driven scheduling alone is not sufficient to build non-trivial real-time systems on top of a microkernel, the operating systems group at TU Dresden developed the quality-assuring scheduling model, a unified model for the admission and scheduling of periodic real-time applications. Although the formal description of the model was widely available and an early benchmark prototype existed, the model was never completely implemented in the Fiasco microkernel.

This thesis focuses on the implementation of the necessary microkernel mechanisms to support the construction of systems employing quality-assuring scheduling on top of Fiasco. I introduce a new infrastructure for scheduling periodic threads, based on reservation timeslices that consist of a time quantum coupled with a priority. User-level schedulers can assign such reservation timeslices to periodic threads and enforce a specific schedule with the help of the microkernel. The kernel can inform user-level schedulers of exceptional scheduling conditions using an asynchronous Preemption-IPC feedback mechanism, which I implemented on top of synchronous L4 IPC.

I present multiple extensions to the existing L4 interface to facilitate the new scheduling infrastructure and introduce absolute timeouts as the time basis for periodic threads. Combined with fine-grained hardware timers these timeouts provide an accurate timing mechanism for user-level applications.

Because threads in microkernel-based systems frequently interact using inter-process communication, I explore how scheduling mechanisms and IPC can be combined to avoid priority inversion. I discuss the requirements for a predictable time-donation mechanism and contrast these requirements with the constraints for the implementation of fast IPC in the Fiasco microkernel.

Organization of this Thesis

This thesis is organized as follows. In Chapter 2, I give an introduction to real-time systems and explain the timing constraints for periodic real-time tasks. I discuss the nature of deadlines and the differences between strictly periodic threads and periodic threads with minimal interrelease times. In Chapter 3, I present the design of the new scheduling infrastructure in the Fiasco microkernel. I then introduce the quality-assuring scheduling model (Section 3.1), propose the decoupling of execution and scheduling contexts (Section 3.2) and examine the redesign of the ready list (Section 3.3). Furthermore, I discuss the new timeout types (Section 3.6) and the Preemption-IPC feedback mechanism (Section 3.8). At the end of the chapter I evaluate how time donation can be used to avoid priority inversion (Section 3.9) and how the helping-lock mechanism in Fiasco can be modified to work correctly in the presence of donation (Section 3.10). In Chapter 4, I present the different scheduling modes (Section 4.1) and describe implementation details for the extended system-call interface (Section 4.2). I evaluate the performance of the new scheduler in Chapter 5 and conclude this thesis with suggestions for future work in Chapter 6.

Chapter 2

Fundamentals and Related Work

2.1 Microkernels

In contrast to monolithic kernels such as Linux [22] or Windows [26], where all the operating-system services, such as memory management, file systems and network stacks, are part of the kernel, microkernels only provide a minimal set of abstractions in the kernel, upon which the remaining functionality can be implemented in user-space. This design methodology facilitates the construction of robust operating systems, because each user-level server executes in its own protected address space, so that a single failing or malicious service does not hamper the remaining system. By restricting the inter-process communication [12] between different servers, microkernel-based systems can provide extra security. Finally, due to their small size, microkernels are easily portable and maintainable and have a reduced memory and cache footprint.

The first generation of microkernels suffered from flexibility and efficiency problems. Despite innovations like external pagers, the Mach microkernel [1] was a refactored Unix kernel with poor performance due to its enormous size. Second-generation microkernels are much smaller in size and provide a fast implementation of inter-process communication. The L4 microkernel interface [19] was originally designed for the x86 processor architecture by Jochen Liedtke and written completely in assembler. L4 kernels have since then been ported to many other architectures. These microkernels only implement address spaces, threads and synchronous IPC between threads in the kernel; the remaining operating-system functionality must be implemented in user-space. Due to licensing problems with the original L4 microkernel, Michael Hohmuth wrote a new microkernel based on the L4 interface. His Fiasco microkernel [8] is fully preemptible and therefore an excellent basis for the construction of real-time systems.

2.2 Real-Time Systems

2.2.1 Time Constraints

In a real-time system every time-critical task has timing constraints, which specify when the task becomes ready to run, how long it executes for and by what time the result must be available. Among the different tasks a precedence relation may exist that restricts the order in which the kernel may execute these tasks.

Figure 2.1 illustrates the time constraints of a real-time task. The *release time* of a task is the time when the task becomes ready for execution. Because tasks contend for resources, including CPU time, the CPU may be busy handling another task at that time, so that the kernel cannot dispatch the new task immediately. Some time after the release time, the scheduler will activate the new task and allocate the CPU to it. I call the point in time when the task starts executing the task's *activation time* and the time the task finishes its work the *completion time*. The time interval between activation time and completion time is the task's *execution time*.

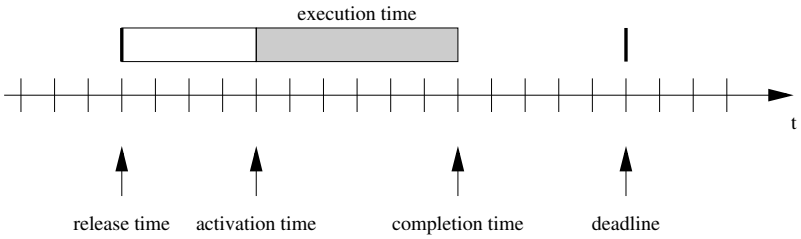


Figure 2.1: Time-Constrained Task

A *deadline*, which is typically associated with each real-time task, specifies when the task's work must be finished before the result decreases in value or becomes useless. The release time of a task is typically associated with a specific event, which may be periodic or aperiodic in nature. The execution time can be fixed or variable, but the deadline of a task is usually fixed.

2.2.2 Deadlines

The deadline of a task can be classified as *hard deadline* or *soft deadline* [24]. Missing a hard deadline usually has fatal consequences for the system, whereas missing a soft deadline is undesirable and degrades the value of the produced result. We can describe deadlines using value functions [13]. A value function is a function of time that describes what value the completion of the task contributes to the overall value of the system. For tasks with a hard deadline the value function immediately drops to $-\infty$ at the deadline; thus a missed deadline causes the entire system to fail. After a soft deadline has passed, the value function either immediately goes to zero or monotonically decreases. A task missing a soft deadline therefore either does not contribute any value to the system, or only a fraction of the possible value had the task met its deadline. Non-real-time tasks can be modelled as having a constant value function without any timing constraints.

2.2.3 Periodic Tasks

Periodically reoccurring computations are called *periodic tasks*. The individually scheduled parts of such a task are often referred to as *jobs*; thus a task is a stream of jobs. The time interval between two consecutive instantiations of a task is called *interrelease time* and the minimum of all interrelease times of a task is the task's *period*. If the interrelease times of all jobs of a task are equal to the task's period, I call the task a *strictly periodic task*, otherwise I call the task a *periodic task with minimal interrelease times*. Tasks whose release times are not known a priori are called *aperiodic task*. If such a task has a hard deadline it is called a *sporadic task*. The work presented in this thesis only focuses on periodic tasks.

Figure 2.2 shows a strictly periodic task with three jobs. The period length of the task is six time units and the execution time of each job is two time units. The task has to wait one time unit in its first period and three time units in its third period after the release time before the kernel dispatches it.

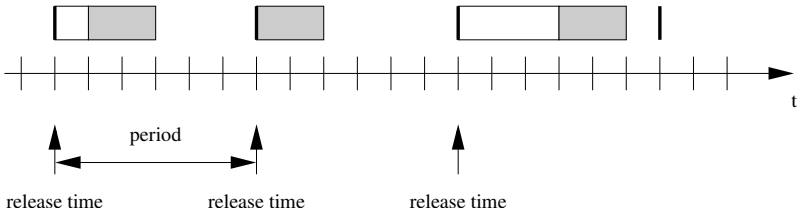


Figure 2.2: Strictly Periodic Task

2.2.4 Priorities

The scheduler assigns each job a positive number that describes the importance of the job. This number is the job's *priority*. A higher number means that the job is more important or more critical than a job with a lower number. Jobs with the same number are of equal importance. Whether a job's priority remains constant or changes over time depends on the algorithm used by the scheduler. We can divide these algorithms into *fixed* and *dynamic* priority scheduling algorithms. The most popular fixed-priority scheduling algorithm is Rate-Monotonic Scheduling [23], which assigns priorities to tasks based on their rate – tasks with a smaller period or a higher rate also receive a higher priority. An example for dynamic priorities is the Earliest-Deadline-First algorithm [3]. It assigns new priorities whenever a job completes or a new job is released. The job with the nearest deadline receives the highest priority. When there are multiple released jobs contending for the CPU, the scheduler in the kernel always gives precedence to the job with the highest priority and dispatches it.

2.2.5 Preemption

When an event causes a higher-priority job to be released, the scheduler may interrupt or *preempt* the currently executing job and allocate the CPU to the high-priority job. Later, when that job has completed its work, the preempted job resumes execution.

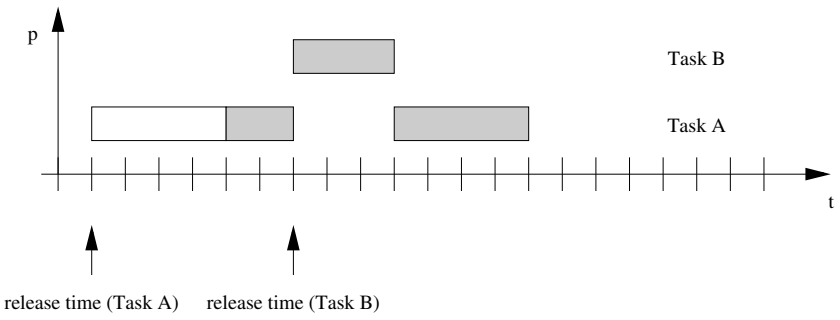


Figure 2.3: Preemption of a Task

Figure 2.3 shows two tasks with different priorities. The low-priority task A has an execution time of six time units, the high-priority task B an execution time of three time units. We see that as soon as B is released, the kernel preempts A and allocates the CPU to B. Once task B has completed, the kernel resumes executing A until completion.

A job that can be preempted and resumed at any time is a *preemptible job*. Jobs scheduled on CPU time can generally be assumed to be preemptible, unless they are executing in a critical section. However, a disk scheduler cannot temporarily suspend read or write requests for a hard disk midway and resume them later. Such disk jobs are therefore *non-preemptible*. When preempting a job, the kernel must save the state of the thread associated with the preempted job, load the state of the thread associated with the preempting job and then switch threads. This action is called *context switch* and typically requires a constant amount of time, which must be taken into consideration when scheduling the jobs.

2.3 Scheduling Algorithms

Scheduling algorithms produce a *schedule* which defines which job the kernel assigns to which CPU at specific time instants, such as the release or completion of a job. At any point in time there is at most one job assigned to each CPU and at most one CPU is assigned to each job. A *valid* schedule executes no job before its release time and honors all precedence constraints among the different jobs. A *feasible* schedule is a valid schedule that additionally satisfies the timing requirements of each tasks, such that no job misses a hard deadline. A set of task is *schedulable*, if there exists at least one feasible schedule.

In reality a task's execution time is usually not known in advance. The execution time depends not only on the jitter in the computation time of the task, but also on hardware effects like cache and TLB misses or pipeline stalls due to mispredicted branches. Although the impact of hardware effects on the execution time of tasks can be diminished using advanced techniques, such as cache coloring [20], many schedulers have to consider the *worst-case execution time* of each task to ensure timely completion of all jobs. Using worst-case execution times for scheduling results in poor resource utilization.

2.3.1 Imprecise Computations

For certain soft real-time applications such as multimedia streaming or tracking the timely computation of an imprecise result is often more valuable than an accurate result produced too late. Therefore imprecise computations [2] trade off result quality to meet computation deadlines. The model splits each task into two parts, a *mandatory* subtask and an *optional* subtask. The mandatory part must be completed before the deadline, because it computes the minimally acceptable result. The optional part refines the result and further improves the quality of the computation. If the system is overloaded, the optional part can be omitted or aborted, so that the result is imprecise. If the optional part runs to completion, the result is said to be precise. This programming model is supported by Flex [21], an object-oriented programming language, which extends C++ constructs with timing constraints.

Chapter 3

Design

3.1 Quality-Assuring Scheduling

The *Quality-Assuring-Scheduling* (QAS) [7] model is a unified model for admission and scheduling, which can be applied to active resources, such as CPU and disk. Similar to imprecise computations, the model decomposes a task into one mandatory part and zero or more optional parts. Quality-assuring scheduling extends imprecise computations by allowing multiple optional parts, and it assumes a given distribution of execution times instead of worst-case execution times. This approach allows us to drastically reduce the amount of reserved resources for applications with soft real-time requirements, thus permitting the system to admit more tasks.

The scheduling model is based on periodic tasks with fixed priorities. Before a task can be admitted, the distribution of its execution time must be known. The task must declare to an admission server which part of its computation is critical and therefore mandatory, and which parts are optional and improve quality. Each part is considered successful only if it is completely executed. The model ensures that the mandatory part always meets its deadline, even in worst-case situations. However, it is sufficient that under system overload, only a user-specified fraction of the optional parts meets the deadline. We call the fraction of completely executed optional parts the task's *quality*. The order of the different parts defines their importance or value. As such, the kernel always executes the task's mandatory part first, then continues with the first optional part and so forth.

Our approach tries to keep the scheduling overhead for periodic tasks low. Periodic tasks are required to pass an admission test before they are started. The computation of the task's scheduling parameters is done outside the kernel in a user-level admission server. If the task passes the admission test, the admission server allocates the CPU to the task for a certain time span called the *reservation time*. For each mandatory or optional part it assigns a tuple of time quantum and priority to the periodic thread that performs the task[1]. I call these tuples *reservation scheduling contexts*, and the thread's normal scheduling parameters *regular scheduling context*.

The calculation of the parameters for reservation scheduling contexts corresponding to mandatory parts uses the worst-case execution time of these mandatory parts. In contrast, the admission server assumes probabilistic execution times for the mandatory parts when admitting the optional parts, thereby overbooking the period of a task with more optional parts than the system could admit using

[1] In the sense of scheduling, the term "task" refers to the computation, not the resource space the thread executes in.

worst-case times. If the mandatory parts consume less time than their worst-case execution time specifies, the system can complete the overbooked optional parts; otherwise the optional parts have to be dropped. Because each mandatory part precedes all optional parts, the admission server assigns each mandatory part of a set of tasks a higher priority than any optional part. It derives priorities for the optional parts using an algorithm we call *Quality Monotonic Scheduling* – an optional part that has a higher quality or a higher value than another optional part also receives a higher priority.

Figure 3.1 illustrates the algorithm for two tasks *A* and *B*. Task *A* has two optional parts and task *B* has three optional parts. At the bottom of the figure the resulting schedule is shown. The jobs O_2 and O_3 of task *B* are overbooked optional parts.

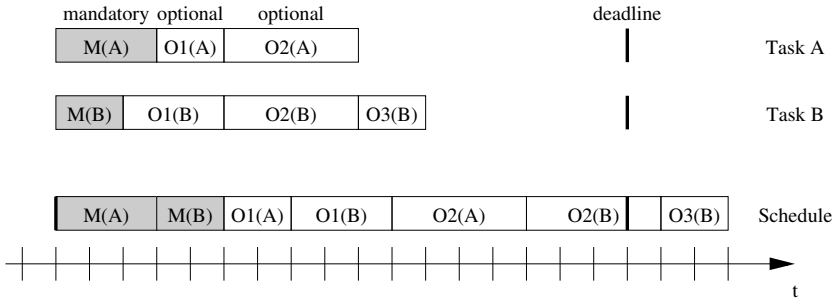


Figure 3.1: Quality-Assuring Scheduling

The admission server also specifies at which time the thread starts its periodic execution. The kernel does not compute a schedule itself. Instead it only provides the mechanisms to enforce a user-defined schedule. We can say that the admission server implements the *scheduling* algorithm and computes the schedule, whereas the kernel performs the *dispatching* of threads according to that schedule.

A periodic thread that owns a reservation scheduling context runs on the priority associated with that scheduling context until it voluntarily releases the reservation scheduling context after finishing the corresponding part of work, or until the time quantum on that scheduling context is depleted. In both cases the kernel assigns new scheduling parameters to the thread using the next reservation scheduling context. If the time quantum on the last reservation scheduling context is depleted, the kernel assigns the scheduling parameters from the thread's regular scheduling context. For periodic threads, the kernel automatically replenishes the time quantum of each of the thread's scheduling contexts and reassigns the scheduling parameters from the first reservation scheduling context at the beginning of each new period.

3.1.1 Interface Requirements

I will now outline the requirements for a scheduler interface to facilitate the implementation of the model in a real-time system.

- The kernel must provide a mechanism to let user-level admission servers specify if a thread is periodic or not.
- For the assignment of reservation scheduling contexts to threads, the kernel must provide an interface which enables admission servers to configure the time quantum and priority associated with such scheduling contexts.

- Admission servers must be able to specify precisely at which point in time a thread shall begin its periodic execution. To achieve the necessary precision, the kernel must export its internal clock to user-level applications and allow applications to use absolute timeouts.

- The kernel must provide a mechanism to stop periodic execution and remove the assigned reservation scheduling contexts afterwards.

- A periodic thread must be able to declare to the kernel when it has finished the part of work corresponding to a particular reservation scheduling context, so that the kernel can activate the next scheduling context for that thread.

- Furthermore, the kernel must implement a mechanism that allows a periodic thread to specify that it has finished its computation in a particular period and that the thread now waits for the beginning of its next period.

- To allow threads to react to exceptional scheduling conditions, for example when a thread misses its periodic deadline or exceeds the time quantum on a reservation scheduling context, the kernel must implement a feedback mechanism to notify the user-level scheduler associated with the thread, which we call the *preempter*.

3.1.2 Programming Model for Periodic Threads

The implementation of quality-assuring scheduling in the Fiasco microkernel provides an easy to use programming model for periodic tasks. Figure 3.2 shows a pseudo-code fragment for a periodic thread on the left and the associated preempter on the right side.

```
periodic thread:

send_admission (request);

while (periodic) {
  rt_next_period();
  do_mandatory();
  rt_next_reservation();
  do_optional_1();
  rt_next_reservation();
  do_optional_2();
  rt_next_reservation();
  ...
  do_optional_n();
}

send_admission (release);
```

```
preempter:

while (running) {
  receive_preemption_ipc (msg);
  switch (msg.type} {
    case TIMESLICE_OVERRUN:
      abort_optional (msg.id);
      break;
    case DEADLINE_MISS:
      adjust_quality();
      break;
  }
}
```

Figure 3.2: Programming Model (Periodic Thread)

Before a thread begins its periodic execution, it negotiates with an admission server and requests that the admission server makes the necessary reservations of resources in the kernel according to the desired quality.

If the admission server admits the thread, it registers the beginning of the thread's first period with the kernel. Once the periodic thread receives a positive notification from its admission server, it is expected to execute an *rt_next_period* call, which signals to the kernel that the thread is waiting for the beginning of the next period. The call blocks the periodic thread until the period begins. When the call returns, the thread can expect to run on its first reservation scheduling context and should start computing the mandatory part of its work. If the thread fails to call into the kernel using *rt_next_period* before its first period begins, the kernel notifies the thread's preempter.

When the periodic thread has finished the part of work for which a particular scheduling context had been reserved, it is expected to perform an *rt_next_reservation* call to release that scheduling context and activate the next one. When the call returns successfully, the kernel has assigned the new scheduling parameters from the next reservation scheduling context and the thread starts consuming the time quantum of that scheduling context. The thread should then start executing its next optional part. The thread then continues computing optional parts, calling *rt_next_reservation* after each of them is complete. Once the thread has completed the last optional part and released the associated reservation scheduling context, it calls *rt_next_period* again, to wait for the beginning of the next period.

The preempter of a thread receives notification messages about scheduling exceptions from the kernel and can react accordingly. The user can configure the preempter to be another thread in the address space of the periodic application, so that it can provide feedback to the application and abort or skip optional parts that run for an excessively long time. Currently the kernel sends two different notification messages. The *timeslice overrun* message signals the preempter that the time quantum on the specified reservation scheduling context was depleted before the thread finished the associated part of work and released it using *rt_next_reservation*. The kernel sends a *deadline miss* message when the thread failed to execute an *rt_next_period* call before its deadline.

3.1.3 Programming Model for Admission Servers

The admission server negotiates with periodic threads according to a user-defined reservation protocol. When it receives an admission request from a periodic thread, it runs an admission test to see if it can find a feasible schedule, so that the mandatory parts of all periodic tasks meet their deadline and all tasks achieve a percentage of completed optional parts according to their specified quality parameter.

Figure 3.3 shows a pseudo-code fragment for a possible admission server. When a periodic thread has been admitted, it is the responsibility of the admission server to configure the scheduling parameters for the thread in the kernel according to the computed schedule. The configured scheduling parameters can include period length, type of periodic task and reservation scheduling contexts. Using the *rt_add* call, the admission server can register a new reservation scheduling context for a periodic thread. The *rt_period* call sets the period length of a thread, while the *rt_begin_periodic* call specifies when a thread's first period begins and whether the thread is a strictly periodic thread or a periodic thread with minimal interrelease times. The calls to configure the scheduling parameters of periodic threads are complemented by *rt_end_periodic*, to stop a thread's periodic execution, and *rt_remove*, to remove all of a thread's reservation scheduling contexts.

```
while (running) {

  receive_admission (caller, reservation);

  switch (reservation.type) {

    case REQUEST:
      admission_test (schedule);
      if (admitted) {
        for (i = 0; i < reservation.parts; i++)
          rt_add (caller, schedule->schedcontext[i]);
        rt_period (caller, schedule->period);
        rt_begin_periodic (caller, schedule->start);
      }
      break;

    case RELEASE:
      rt_end_periodic (caller);
      rt_remove (caller);
      break;
  }

  notify (caller);
}
```

Figure 3.3: Programming Model (Admission Server)

3.2 Scheduling Context

The L4 microkernel interface achieves fast message transfers during IPC because the sender of an IPC donates its time quantum and priority to the receiver of the IPC, thereby avoiding an invocation of the scheduler. The kernel therefore needs to provide a mechanism that allows a thread to switch to another thread, without consuming the other thread's time. Because an admission server can assign multiple tuples of time and priority to a periodic thread and to better facilitate the donation of time from one thread to another, I completely decoupled execution and scheduling. I factored out scheduling and accounting parameters and put them into a new class called *Sched_context*. The class that implements execution contexts is called *Context*. Scheduling contexts are schedulable entities and belong to exactly one execution context, such as a thread. Each execution context aggregates one scheduling context – the regular scheduling context – which becomes part of the thread's TCB. Additional reservation scheduling contexts can be allocated to each periodic thread via a slab allocator. Figure 3.4 shows how the kernel maintains the scheduling contexts of a thread in a doubly-linked circular list; the regular scheduling context is a dedicated entry point into that list, because its offset in the TCB is known.

During regular thread switching the kernel switches to the execution context *and* the scheduling context of the destination thread. When a thread donates its time and priority to another thread,

the kernel only switches to the destination thread's execution context, so that the scheduling context remains the same. The destination thread then consumes the time on the donated scheduling context instead of consuming its own time. This time donation mechanism is comparable to the migrating threads model [6], except that time donation migrates scheduling parameters (time quantum, priority) instead of migrating executable entities.

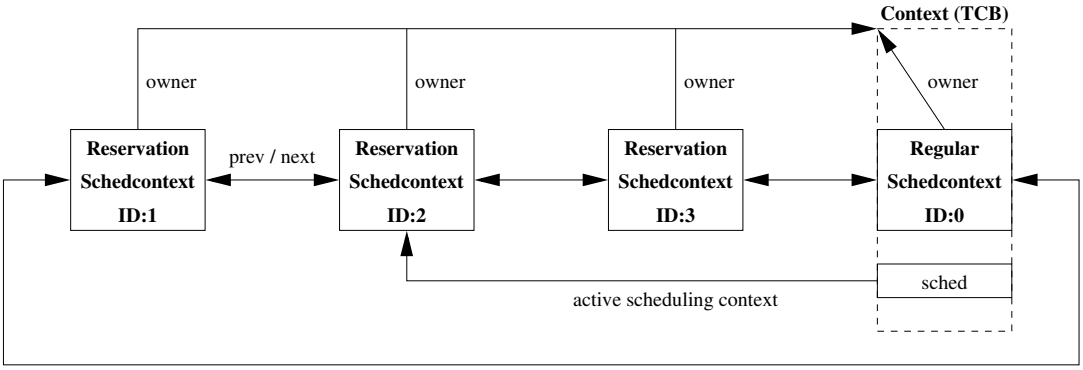


Figure 3.4: Scheduling Contexts

A scheduling context implements a time quantum coupled with a priority and has the following attributes:

- **Owner**: The owner of a scheduling context is a pointer to the execution context (thread) to which this scheduling context belongs. The owner pointer is necessary for additional reservation scheduling contexts, because computing the corresponding owner thread from their addresses is impossible – they are allocated from a slab allocator at the time of their creation and reside in the kernel's slab space. The kernel sets the owner pointer for all scheduling contexts, including the regular scheduling context. The owner pointer is not strictly necessary for the regular scheduling context, because each regular scheduling context is aggregated in the TCB of the owning thread. Therefore, the owner could be easily computed by doing simple address rounding arithmetics.

- **ID**: Scheduling contexts are assigned positive numbers in order of their creation, starting at 0. These identification numbers are local to the execution context, which owns the scheduling context. The regular scheduling context is always assigned ID 0, because it is constructed at thread creation time. The kernel assigns identification numbers consecutively, starting from 1, to newly created reservation scheduling contexts. These IDs are part of Preemption-IPC messages sent to the owning thread's preempter and allow user applications to attribute preemption events to the correct scheduling contexts.

- **Prio**: Priority associated with this scheduling context. It is important to note that the priority is an attribute of a scheduling context. Execution contexts (threads) have no associated priority. The priority of a thread is always determined by the priority of the scheduling context on which the thread executes.

- **Quantum**: The total time quantum associated with this scheduling context. The quantum is specified in microseconds. The kernel rounds the user-specified value up to the nearest multiple of the granularity supported by the kernel's internal timer source.

- **Left**: Remainder of the original time quantum, also in microseconds. Whenever a scheduling context is preempted, the kernel saves the remaining time quantum in this variable. If the remaining time quantum reaches zero, the scheduling context is refilled to the total quantum.
- **Preemption Time**: Point in time at which the most recent preemption event occurred on this scheduling context (see Section 3.8).
- **Preemption Type**: Type of preemption event that occurred (see Section 3.8).
- **Prev, Next**: Pointers to the previous and next scheduling contexts in the thread's list of scheduling contexts.

For the understanding of the following sections it is necessary to define the meaning of the following terms.

- As shown in Figure 3.4, a thread can have multiple scheduling contexts, which the kernel maintains in a doubly-linked list. I call this list the thread's **list of scheduling contexts**. The accessor function returning the regular scheduling context of thread T in the implementation is `T->sched_context()`. All other scheduling contexts in the list can be found by traversing the list starting at the regular scheduling context.
- The scheduling context in the list, which the thread currently uses or which it will use next and according to whose priority the thread is enqueued in the ready list, if it is ready, is called the thread's **active scheduling context**. The accessor function returning the active scheduling context of thread T in the implementation is `T->sched()`. All other scheduling contexts of the thread are inactive, but may become the thread's active scheduling context at some later time.
- Among all ready threads, the scheduler selects the active scheduling context with the highest priority and dispatches it. That scheduling context becomes the **current scheduling context** and the kernel allocates the CPU to it for the duration of its time quantum. The function returning the current scheduling context in the implementation is `current_sched()`. Other active scheduling contexts may become the current scheduling context at some later time.
- Once the scheduler has selected a scheduling context to become the current scheduling context, it selects a thread to run. That thread is either the owner of the current scheduling context or a different thread to which the owner has donated the scheduling context. The selected thread becomes the **current thread** on the CPU. The function returning the current thread in the implementation is `current()`. Other ready threads may become the current thread when their active scheduling context becomes the current scheduling context or when the current scheduling context is donated to them.

In the absence of donation, the following rules apply:

- The current thread's active scheduling context is the current scheduling context:
  `current()->sched() == current_sched()`
- The owner of the current scheduling context is the current thread:
  `current_sched()->owner() == current()`.

When a scheduling context has been donated, the preceding two rules are no longer valid, but the following rule still is:

- The current scheduling context is the active scheduling context of its owner: `current_sched()->owner()->sched() == current_sched()`.

3.3 Ready List

The kernel usually invokes the scheduler whenever it is unclear which thread should be dispatched next. The scheduler is then responsible for choosing both the scheduling context on which the next thread should run, as well as the next thread itself. To be able to make such scheduling decisions quickly, the kernel keeps a list of runnable threads – the ready list.

The Fiasco microkernel supports 256 priorities ranging from 0 to 255. If multiple scheduling contexts have the same priority, the kernel schedules them in a round-robin manner. For each thread there is exactly one of its own scheduling contexts active at any time. If the thread is ready the kernel enqueues it according to the priority of its own active scheduling context.

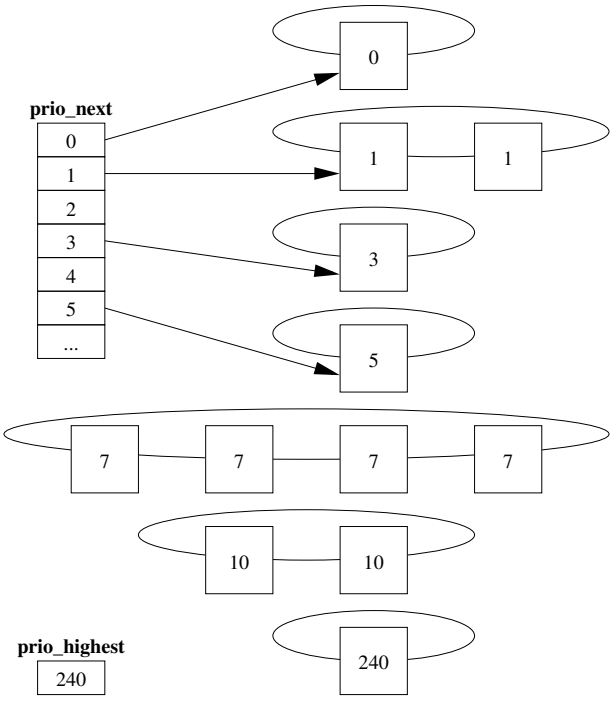


Figure 3.5: Structure of the Ready List

Figure 3.5 shows an example ready list with twelve threads currently enqueued. For each of the 256 priority levels there exists a cyclic linked list of threads at that priority. The priority rings are formed using two pointers in each thread's TCB, `ready_prev` and `ready_next`. These pointers point to the sibling threads in the same ring. If a thread is the only thread at its priority level, both pointers point to the thread itself. If a thread is not enqueued in the ready list, its `ready_next` pointer is a null pointer and its `ready_prev` pointer is stale. To quickly find the entry point of each priority ring, the

kernel keeps an array `prio_next` of pointers to the thread of each priority level that should run next. If a priority level is empty, the corresponding `prio_next` slot contains a null pointer. The kernel additionally stores the number of the highest populated priority level in the `prio_highest` variable.

The decision to implement the ready list by inter-linking threads instead of scheduling contexts stems from performance considerations. If the ready list were implemented by linking scheduling contexts and the scheduler traversed the rings to select the next thread for dispatching, the scheduler would have to dereference the owner pointer of each scheduling context to determine the owner thread. Due to an invariant in the ready list, the owner's state must be checked to verify that the thread is still ready to run. The dereference operation is likely to cause additional TLB and cache misses, which the current implementation avoids.

3.3.1 Lazy Queueing

In a typical RPC scenario a client sends a request to a server and then blocks, waiting for a reply. The server usually blocks until a request comes in, then handles the request, sends back a reply and blocks again. If we demand that the ready list be always up to date and that only ready threads are enqueued in the list, then the preceding RPC scenario requires multiple updates of the list:

- After sending the request, the client is no longer ready and must be dequeued.
- Upon receiving the request, the server becomes ready and must be enqueued.
- After sending back the reply, the server is no longer ready and must be dequeued.
- Upon receiving the reply, the client is once again ready and must be enqueued.

This solution is far from optimal and results in heavily degraded IPC performance. A better solution has been proposed in [18] and is implemented in all recent L4 kernels. The ready list can be viewed as a cache in which the scheduler can quickly find each ready thread at any time. One thread is per definition always ready – the thread that is currently running on the CPU. The existence of the idle thread ensures that there is always at least one ready thread on each CPU. The key idea is to introduce the following invariant:

All ready threads must be enqueued in the ready list, except for the currently executing thread. Threads that are not ready are not removed from the ready list at the moment they block, but the next time the scheduler traverses the ready list.

If we apply the invariant to the aforementioned RPC scenario, IPC can be implemented without any ready list manipulation. The server need not be enqueued while it executes, because during that time it is the current thread on the CPU, and the client need not be dequeued and reenqueued. To keep track of all ready threads, this performance optimization implies that the kernel must enqueue the current thread when switching away from it or when a consistent view of the ready list is required, for example when making scheduling decisions.

3.3.2 Enqueue Operation

As I described in Section 3.3, the ready list consists of threads linked together in multiple rings. Modifying the ready list is a critical section and mutual exclusion must be guaranteed. The current

uniprocessor implementation takes the CPU lock, which effectively disables interrupts on the local CPU. For SMP systems the kernel should use one ready list local to each CPU. The `ready_enqueue` function first checks if a thread is already in the ready list or if it is not ready. In either case the function returns immediately. Otherwise the function determines the correct priority ring into which the thread should be enqueued. The current priority of the thread is always the priority of the thread's active scheduling context: `prio = sched()->prio()`.

Recall from the description of the invariant in Section 3.3.1 that the current thread need not be enqueued in the ready list all the time, even though it is ready. Therefore special care must be taken when enqueueing a thread with the same priority as the current thread. Such a scenario occurs when the current thread is not enqueued in the ready list, because of the invariant, and an asynchronous event, such as the expiration of a timeout, wakes up a different thread with the same priority as the current thread. To maintain the correct order of threads for round-robin scheduling, the current thread must be enqueued first, because it was ready first. For that reason the enqueue function checks the current thread's priority and enqueues the current thread first, if necessary. Now the woken-up thread itself can be enqueued. If its priority is greater than `prio_highest`, then `prio_highest` must be adapted to the thread's priority. Finally the thread can be inserted into the corresponding priority ring. If the priority ring was previously empty, the thread immediately becomes the next thread to run at that priority level and forms a ring consisting only of itself. If other threads already exist at that priority level, the kernel inserts the thread at the end of the priority ring by enqueueing it before the next thread that should run at that priority.

3.3.3 Dequeue Operation

To dequeue a thread from the ready list, the kernel also acquires the CPU lock for the duration of the operation. If the thread is not in the ready list – its `ready_next` pointer is a null pointer – the function returns immediately. The kernel locates the thread's priority ring via the thread's current priority – the priority of the thread's currently active scheduling context. It is therefore important that whenever a thread's priority changes the thread must be dequeued from its old priority ring and enqueued in its new priority ring. If this rule is violated the `ready_dequeue` function updates the wrong ring and the ready list becomes corrupt. If the thread happens to be the next thread to run at its priority, the kernel logically rotates the priority ring so that the thread following the thread to be dequeued is now the next thread to run. If no other thread exists at that priority level, the ring becomes empty. The actual dequeue operation removes the thread from its ring by linking the thread's predecessor and successor together and making the thread's own `ready_next` pointer a null pointer to mark it dequeued. Finally the kernel must check if the dequeued thread was the last thread of the highest priority level. In that case `prio_highest` must be recomputed by stepping through the `prio_next` array until a nonempty ring with a lower priority is found. The search finally terminates at the idle thread, which has the lowest priority in the system.

3.4 Timeslice Switching

A thread can donate time to another thread by instructing the kernel to switch to the other thread without also switching to that thread's active scheduling context. The other thread then effectively executes on an active scheduling context, which it does not own. Donation can be transitive; that is, threads can also donate active scheduling contexts which they received by means of donation from

other threads and which they do not own. In the presence of donation the kernel can never assume that the current thread is the owner of the current scheduling context. However, it can determine the owner by inspecting the owner attribute of the current scheduling context. If the current thread has consumed all remaining time of the current scheduling context, the kernel preempts the current thread and invokes the scheduler. The scheduler deactivates the active scheduling context of the owner and the owner's next scheduling context becomes the owner's new active scheduling context before the scheduler selects a new current scheduling context.

If the expired scheduling context was a reservation scheduling context, the kernel activates the next scheduling context in the thread's list of scheduling contexts. This can be either another reservation scheduling context or the thread's regular scheduling context. If the thread exhausted the owner's regular scheduling context, the kernel replenishes that scheduling context using the value of the total time quantum. Timeslice switching not only happens when a scheduling context runs out of time. Other possibilities include threads voluntarily releasing the CPU by yielding their active scheduling context or threads starting their next period.

In contrast to older versions of the Fiasco microkernel, which had the timeslice switching code duplicated in multiple functions, I have implemented a new function `Context::switch_sched()`. The kernel always calls this function in the context of the thread owning the scheduling context to be deactivated. The function's only parameter is the owner's next scheduling context to be activated. If the deactivated scheduling context was the current scheduling context, then the function invalidates the current scheduling context. When the scheduler runs the next time it selects a new current scheduling context. The kernel always replenishes the remaining time quantum of a scheduling context which it deactivates to the value of the total time quantum. If the kernel were to replenish scheduling contexts upon activation instead, it would have to distinguish between scheduling contexts that still have time remaining and must not be replenished and scheduling contexts that have run out of time.

In case of reservation scheduling contexts, the deactivated and the activated scheduling contexts typically have different priorities. Otherwise they could be set up as one scheduling context with a time quantum equal to the sum of both time quanta. However, there are also cases when the priority remains the same, for example when a thread's regular scheduling context expires and the kernel replenishes and reactivates it. The kernel must update the ready list according to the changed priorities, if necessary. The following situations require an update of the ready list:

- If the owner is not enqueued in the ready list, it is either not ready or it is the current thread, or it is ready but its scheduling context, which was just deactivated, has a total time quantum of zero. In the latter case, if the newly activated scheduling context has a nonzero total time quantum, then the kernel must enqueue the owner in the ready list.
- If the owner is already enqueued in the ready list, irrespective of the fact whether it is ready or not, and the priority changes, the kernel must dequeue the owner from its old priority ring and reenqueue it according to the new priority. The enqueue operation moves the owner to the end of the new priority ring.
- If the priority does not change, the owner must nevertheless be moved to the end of its priority ring to ensure round-robin fairness. To accomplish that, the kernel must typically dequeue and reenqueue the owner. However, there is one scenario which can be optimized. If the owner is the next thread to run at its priority, then the kernel can simply rotate the priority ring by setting `prio_next` accordingly. The exception is the ring for priority 0. The next thread to run at priority 0 is always the idle thread, and even if the idle thread's time quantum expires, the

kernel never rotates that ring. This ensures that the scheduler never dispatches any thread at priority 0, except for the idle thread. Setting a thread's priority to 0 thus removes the thread's ability to run using its own time. The only possibility for a priority 0 thread to run is another thread donating a scheduling context with a nonzero priority.

3.5 Timer Interrupt

For the implementation of the kernel clock and for timeouts, the Fiasco microkernel must provide an accurate hardware timer source to drive the kernel clock and to support the implementation of fine-grained timeouts. Recent x86 platforms provides multiple hardware timer sources to choose from. Figure 3.6 lists the period and interrupt vector for each timer source currently supported in the Fiasco microkernel.

Timer Source	Vector	Period	Granularity	Mode
PIT	0x20	1000 μs	1000 μs	Periodic
RTC	0x28	976 μs	976 μs	Periodic
APIC	0x3d	1000 μs	1000 μs	Periodic
APIC	0x3d	–	1 μs	One-Shot

Figure 3.6: Hardware Timers

Unless operating in one-shot mode, the hardware timer periodically generates an interrupt, which causes the current thread to enter the kernel through the corresponding interrupt gate. After acknowledging the interrupt at the PIC[2], the kernel updates its internal 64 bit clock by adding the number of microseconds that have elapsed since the last timer interrupt. Finally the timer interrupt handler processes the list of active timeouts, checking if any of them expired and invoking the scheduler if necessary.

For periodic timer interrupts, the kernel programs the hardware timer once at bootstrap time. When the timer expires, it automatically rearms itself. However, the granularity of the kernel clock and the timeouts is limited to the granularity of the hardware timer providing the interrupts. Decreasing the period and thus improving the granularity of a periodic timer results in more timer interrupts, which unnecessarily slow down the system. Ideally the hardware timer should only generate timer interrupts when a specific event occurs, for example when a timeout expires.

This can be accomplished by using one-shot timer interrupts. In contrast to periodic timers, a one-shot timer does not rearm itself. Whenever a timer interrupt occurs, the kernel computes the time of the nearest event in the future and reprograms the next timer interrupt accordingly. To easily determine the nearest event in the future I mapped all timing-relevant events to timeouts as described in Section 3.6.

Using one-shot timers the kernel can provide a timer granularity in the microsecond range, but that is only feasible if the hardware timer can be reprogrammed without excessive cost. Fortunately the APIC timer can be reprogrammed in less than 100 cycles.

[2] Programmable Interrupt Controller

One-shot timer interrupts do not necessarily improve system performance. If there are many timeouts in a short period of time, then it may be more feasible to handle them with one periodic timer interrupt instead of using a one-shot timer interrupt for each of them, at the cost of losing precision.

I measured the overhead of timer interrupts depending on the hardware timer used by the kernel and depending on the number of expired timeouts and present the results in Section 5.2 of this thesis.

3.6 Timeouts

To protect against stalled IPC when communicating with unresponsive or malicious parties, the existing L4 interface supports IPC timeouts. Users can specify different timeouts for the send and receive phases, and for the handling of page-faults that can occur during Long-IPC. Traditionally these IPC timeouts were the only type of timeout supported in the Fiasco microkernel. The implementation of quality-assuring scheduling in the kernel required the addition of new timeout types to detect deadline misses of periodic threads and to recognize the depletion of the time quantum of the current scheduling context. This section gives a brief overview of the different timeout types.

The kernel keeps all timeouts in a doubly-linked list, sorted by ascending wakeup time. The wakeup time is an absolute 64-bit time value, measured in microseconds from the system bootstrap, representing the point in time at which the timeout is supposed to trigger. Because the nearest timeout is always the head of the timeout list, the kernel can easily check if a one-shot timer needs to be reprogrammed. If the head of the timeout list is dequeued or a new head is prepended to the list, the kernel must reprogram the APIC timer according to the wakeup time of the new head. If a timeout is added or deleted in the middle or at the end of the list, the APIC need not be reprogrammed. If the kernel uses a periodic timer, then the timer never needs to be reprogrammed.

Upon each timer interrupt, the kernel invokes the `do_timeouts()` function, which compares the wakeup time of the head of the timeout list to the kernel's internal clock. If the clock value is greater than or equal to the timeout's wakeup time, then the timeout has expired. The kernel dequeues an expired timeout from the list and invokes the timeout's `expired()` function, then proceeds with the next timeout until it finds a timeout that has a wakeup time in the future. Due to the sorting of the timeout list, all following timeouts also have future wakeup times and need not be checked.

Each `expired()` function has a boolean return value, which when true signals that the timeout has woken up a thread with a higher priority than the current thread, which had been interrupted by the timer interrupt. It is not possible to immediately switch to the woken up thread if it has a higher priority, because other timeouts that also just expired can potentially wake up a thread with an even higher priority. The `do_timeouts()` function therefore defers calling the scheduler until it has processed all expired timeouts and invokes the scheduler only if at least one timeout signalled a reschedule request.

The generic `Timeout` class implements an abstract timeout type and all the list handling for the timeout list, with functions to enqueue and dequeue timeouts. It also declares the pure virtual function `expired()`, which all derived timeout classes must overwrite. The timeout classes `Timeslice_timeout`, `IPC_timeout` and `Deadline_timeout` are derived from the generic Timeout class and differ from each other in the action performed when the timeout expires and the kernel object associated with the timeout. Figure 3.7 shows a class diagram of the relationship between the different timeout classes.

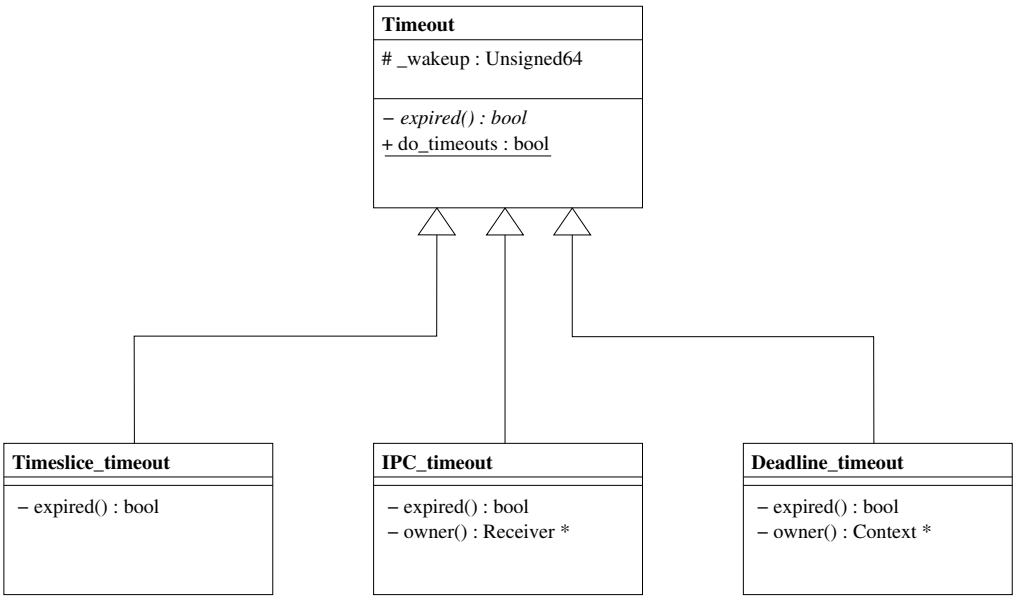


Figure 3.7: Timeout Classes

3.6.1 IPC Timeouts

The kernel uses an IPC timeout to abort an ongoing IPC for the thread that installed the timeout during an IPC operation. The kernel employs its knowledge of the memory location of an IPC timeout to determine the owner. Because the IPC code allocates send- and receive timeouts on the kernel stack of the thread invoking the IPC operation, and because kernel stacks and TCBs are adjacent in the kernel's virtual memory area, the owner can be easily computed by rounding down the timeout's address to the nearest multiple TCB size. Using this arithmetic trick, we can save 4 bytes for the owner pointer for each timeout. The `expired()` function for the IPC timeout, which is called when the timeout's wakeup time has been reached, removes the owner's `Thread_ipc_in_progress` flag to abort IPC, wakes the thread up by setting the `Thread_ready` flag and enqueues the thread in the ready list.

3.6.2 Deadline Timeouts

Because the quality-assuring scheduling model supports periodic threads, the kernel needs a mechanism to implement different periods for each such thread. For this reason I added a deadline timeout to each thread control block. An expired deadline timeout signals the end of the current period for the owner thread. For strictly periodic threads the end of one period is also the beginning of the next period. The kernel uses a thread's deadline timeout for the following purposes:

- During a thread's transition from conventional to periodic mode, the deadline timeout specifies the point in time the thread's first period commences. I explain the scheduling mode transition in Section 4.1.

- While a thread is executing in periodic mode, a deadline timeout is active to detect a deadline miss of the thread.

- A thread's expired deadline timeout also indicates the possibility of starting the thread's next period.

Similar to IPC timeouts, the owner of a deadline timeout can be easily computed using arithmetic on the timeout's address, because the deadline timeout is aggregated in the owner's TCB. When a deadline timeout expires, the kernel checks if the owner thread is ready to begin the next period. A thread signals that it is ready to start the next period by performing an IPC with next_period semantics. Such an IPC puts the thread into a special "waiting for next period" state. For a detailed description of next_period IPCs, see Section 4.5. The `expired()` function checks if the thread is in the "waiting for next period" state and if it is not, the kernel synthesizes a "Deadline Miss" Preemption-IPC message and sends it on behalf of the thread to the thread's preempter. If the thread had already been waiting for the beginning of the next period and all conditions for the beginning of a new period are satisfied, the kernel wakes the thread up and arranges for the next period to begin.

3.6.3 Timeslice Timeouts

When the scheduler activates a scheduling context as current scheduling context, it programs a timeslice timeout that is triggered when the time quantum of that scheduling context runs out. The kernel calculates the wakeup time of the timeslice timeout as the kernel's internal clock plus the remaining time quantum of the activated scheduling context. Because there can be only one scheduling context active on a CPU at any time, the number of timeslice timeouts is bounded by the number of CPUs in the system. Upon expiration of the timeslice timeout, the active scheduling context has no time left and the current scheduling context becomes invalid. The action performed by the `expired()` function differs depending on the type of scheduling context. In case of a regular scheduling context, indicated by the ID 0, the kernel replenishes it to the total quantum and moves the thread to the end of its priority ring. When a reservation scheduling context runs out of time, the kernel replenishes it to the total quantum and activates the next scheduling context in the thread's list of scheduling contexts. Additionally the kernel synthesizes a "Timeslice Overrun" Preemption-IPC message and sends it on behalf of the thread that owns the scheduling context to that thread's preempter. When the current scheduling context expires, a reschedule is always required to select a new current scheduling context and a thread to run on it.

3.7 Absolute Timeouts

The L4 reference manual [19] defines the format of a timeout descriptor for relative timeouts with a mantissa m and an exponent e as shown in Figure 3.8.

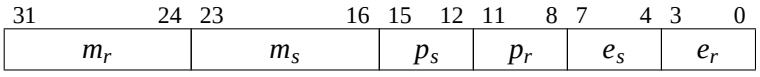


Figure 3.8: Timeout-Descriptor Format

Except for for the special cases $t_{rel} = \infty$ ($e = 0$) and $t_{rel} = 0$ ($e \neq 0,\ m = 0$), the kernel calculates the relative send- and receive timeout values from m and e using the formula $t_{rel} = m \cdot 4^{15-e}\ \mu s$ and then internally transforms them into absolute 64 bit wakeup times by adding the kernel clock

value: $wakeup = clock + t_{rel}$. For absolute timeouts user applications compute the wakeup time themselves using the kernel-provided clock field in the kernel info page and then pass the wakeup time to the kernel. The kernel synchronizes the clock field in the kernel info page with the hardware timer used for the computation of wakeup times inside the kernel. The granularity of the user-visible clock therefore depends on the granularity of the hardware timer in the kernel as shown in Figure 3.6.

The fields p_s and p_r describe timeouts for the send and receive phase of page-fault IPC and are not of interest here.

Because user applications typically use timeouts in combination with the *ipc* system call, I had to find a solution that does not use any extra register space for absolute timeouts. The ABI for the *ipc* system call does not leave any free registers, so there are only two ways to implement absolute timeouts:

- User applications could pass the wakeup time on the stack of the thread invoking the *ipc* system call. Using the stack would allow the ABI to pass arbitrarily large amounts of data, so that the wakeup times could be passed as complete 64 bit time values without the need to encode them in the application and decode them again in the kernel. However, using the stack results in extra memory accesses, extra cache and TLB misses and, worst of all, there is the possibility of the kernel page-faulting on the user stack. Forcing the user to ensure the stack is always mapped, and the extra penalties resulting from memory accesses did not seem the best idea to me.

- The second possibility was to reuse the timeout descriptor used for relative timeouts, because the ABI allocates a register for it, thereby avoiding the problems with the previous option. However, the timeout descriptor only provides very limited space for the description of one timeout (8 bit mantissa, 4 bit exponent), so that I had to develop an encoding to shrink a 64 bit wakeup time to 12 bit. Furthermore the kernel must be able to identify if a timeout descriptor as specified by a user application is to be interpreted as relative timeout or absolute wakeup time.

To encode an absolute timeout's wakeup time such that it fits into the 12 bits of a timeout field in the timeout descriptor requires the user to omit some less important bits of the wakeup time's 64 bit representation:

- The most significant bits of a 64 bit wakeup time typically do not change between the time the user application encodes the wakeup time and the time the kernel decodes it. Omitting them results in the wakeup time becoming ambiguous when the omitted bits do start changing, so that absolute timeouts have a limited validity interval.

- A certain number of the least significant bits of a 64 bit wakeup time can be omitted, thereby lowering the timeout's granularity. The user should be able to specify what granularity he or she wants to achieve.

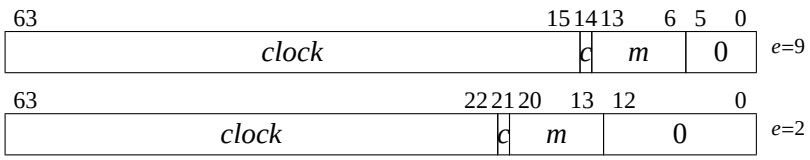


Figure 3.9: Wakeup-Time Representation

Two example encodings are shown in Figure 3.9. User applications choose a timeout granularity that defines how many of the wakeup time's least significant bits are assumed to be zero and therefore need not be passed to the kernel. The exponent e specifies that $15 - e$ of the wakeup time's least significant bits are zero bits. The following 9 bits are passed to the kernel as 8 bit mantissa m and clock bit c. The remaining most significant bits are assumed to not change and are therefore not passed to the kernel either. The number of the omitted most significant bits is $64 - (15 - e) - 9$ or $40 + e$. The exponent e therefore defines the timeout's granularity as 2^{15-e} μs and the timeout's validity interval as 2^{23-e} μs. The mantissa m divides the validity interval into 256 time chunks, each with a size equal to the granularity.

e_s, e_r	relative timeout range			absolute timeout granularity	absolute timeout validity
0		∞		∞	∞
1	268.43 s	...	19.01 h	16.38 ms	4.19 s
2	67.10 s	...	4.75 h	8.19 ms	2.09 s
3	16.77 s	...	71.30 m	4.09 ms	1.04 s
4	4.19 s	...	17.82 m	2.04 ms	524.28 ms
5	1.04 s	...	4.45 m	1.02 ms	262.14 ms
6	262.14 ms	...	66.84 s	512 μs	131.07 ms
7	65.53 ms	...	16.71 s	256 μs	65.53 ms
8	16.38 ms	...	4.17 s	128 μs	32.76 ms
9	4.09 ms	...	1.04 s	64 μs	16.38 ms
10	1.02 ms	...	261.12 ms	32 μs	8.19 ms
11	256 μs	...	65.28 ms	16 μs	4.09 ms
12	64 μs	...	16.32 ms	8 μs	2.04 ms
13	16 μs	...	4.08 ms	4 μs	1.02 ms
14	4 μs	...	1.02 ms	2 μs	512 μs
15	1 μs	...	255 μs	1 μs	256 μs

Figure 3.10: Timeout Ranges

Figure 3.10 gives an overview over the timeout ranges for relative timeouts and the granularity and validity intervals for absolute timeouts when using different values of e. The case $e = 0$ always describes an infinite timeout. The case $m = 0$ only describes a zero timeout for relative timeouts but not for absolute timeouts.

Let us now examine how the kernel recomputes the 64 bit wakeup time from the timeout descriptor passed to it by the user; that is the 4 bit exponent e, the 8 bit mantissa m and the clock bit c. The clock bit extends the mantissa from 8 to 9 bits, thus doubling the length of the interval, which user applications can use (*User Validity Interval*) for the kernel (*Double Kernel Interval*).

Figure 3.11 shows two examples. Each label *Computation* describes the point in time when the user computes the wakeup time. The wakeup time must be no later than *Latest Wakeup*. The arrow labelled *Recomputation* describes the point in time at which the kernel recomputes the wakeup time using the $41 + e$ most significant bits of the kernel clock, the 8 bits of the user-supplied mantissa and $15 - e$ least significant zero bits. The kernel then compares the clock bit computed and passed by the user with the clock bit of the kernel's recomputed wakeup time. If both differ the wakeup time as computed by the

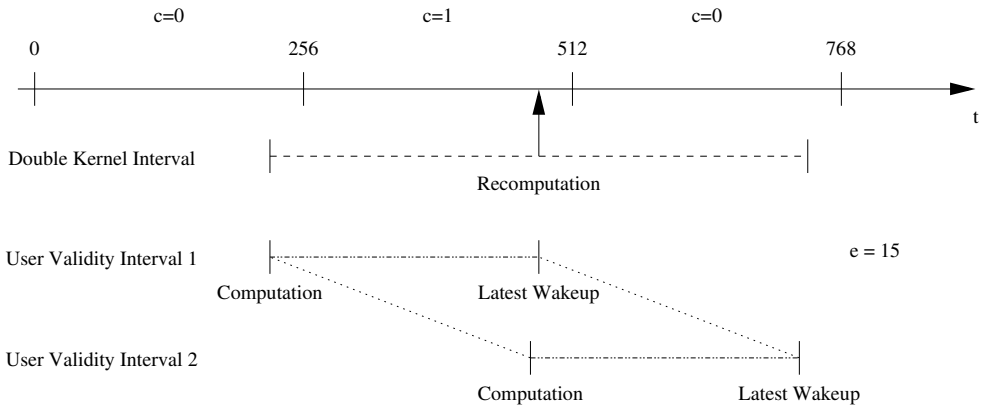


Figure 3.11: Timeout Recomputation

kernel is off by one validity interval and the kernel either adds or substracts $2^{23-e}\ \mu s$ to get the correct value.

For absolute timeouts to work correctly, user applications must ensure that the time intervals between *Computation* and *Recomputation* and between *Computation* and the wakeup time are both no longer than the validity interval.

3.8 Preemption IPC

Preemption IPC is a kernel mechanism to notify a thread's preempter about scheduling exceptions related to the thread itself or related to one of the thread's reservation scheduling contexts. The kernel sends two different types of Preemption IPC, which were already mentioned in Sections 3.6.2 and 3.6.3. This section explains the circumstances that lead to a Preemption IPC being sent, the format of Preemption-IPC messages and the queueing of these messages in the kernel. In the current implementation the kernel generates the following two types of Preemption IPC:

- **Timeslice Overrun**: When a reservation scheduling context expires before the thread running on that scheduling context has voluntarily released it, the kernel notifies the preempter of the thread that owns the scheduling context. The kernel detects a timeslice overrun when the CPU's timeslice timeout expires. If the kernel preempts a thread on a regular scheduling context, it sends no Preemption IPC, because the kernel simply replenishes that scheduling context and does not switch to a different scheduling context.
- **Deadline Miss**: When a thread's next period begins and the thread did not execute a next-period IPC in time, to declare that it is waiting for the beginning of the next period, the kernel sends a Preemption IPC to the thread's preempter. The kernel detects a deadline miss when a thread's deadline timeout expires and the thread is not in the "waiting for next period" state.

Both preemption events are triggered by timeouts and are thus asynchronous to the normal execution of a thread. A preempter must be able to attribute a Preemption IPC it receives to the thread for which the kernel generated it.

For the sending of Preemption-IPC messages I evaluated the following three models:

- **Time-Fault**: We can consider the preemption of a thread by the kernel a resource fault. The preempted thread is lacking time to continue to run. Similar to page-faults, we could model the preemption of a thread as a time-fault. The kernel could synthesize a time-fault IPC message on behalf of the preempted thread and send it to the thread's preempter (time pager), which would then have to reply with a new time quantum for the preempted thread. As with page-faults, masquerading as the faulting thread would require that the kernel use the thread's sender role. However, if the thread was already engaged in a regular IPC at the time of the fault, the kernel would have to suspend the ongoing regular IPC and nest the time-fault IPC inside it. It follows that for the duration of the time-fault IPC, the regular IPC would be stalled. Unlike page-faults, where the faulting thread cannot execute until the pager responds with a mapping to resolve the fault, a thread that generated a time-fault could also receive additional time from other threads by means of donation. However, if the faulting thread were handling a time-fault IPC at the time another thread attempted a rendezvous for a regular IPC with time donation, it could not accept the regular IPC because it would already handling a time-fault IPC. Another problematic aspect is the time that passes from the occurrence of the time-fault until the preempter responds. This time counts against the faulting thread's deadline. In theory it is possible that the handling of a deadline-miss time-fault IPC for a thread would cause the thread to miss its next deadline, thus leading to a cascading effect of time-faults and deadline misses. I decided that this solution is not viable for Preemption IPC, because it stalls the faulting thread.

- **Virtual Interrupt**: The fundamental problem with time-faults is that they stall the execution of the faulting thread for the duration of the Preemption-IPC handling. As an alternative I tried a more asynchronous approach and modelled Preemption IPC as a virtual interrupt. When the kernel preempts a thread, it triggers a virtual interrupt and attempts to send the Preemption-IPC message to a preempter attached to that virtual interrupt. This model is similar to IPC messages being generated for hardware interrupts. To avoid losing Preemption-IPC messages when the preempter is not ready to receive, the kernel must be able to queue such messages. Instead of allocating extra memory resources for queueing, my solution stores a pending preemption event inside the scheduling context for which the kernel generated it. To support multiple preempters the kernel has to either provide one virtual interrupt for each preempter or must allow multiple preempters to attach to a single virtual interrupt. The latter option would require the kernel to enqueue in the sender list of each preempter for which there is a pending preemption event; however, the current L4 interface does not allow multiple concurrent IPC partners to attempt a rendezvous with. The other option – using one virtual interrupt for each preempter, requires the kernel to maintain an association between preempter threads and virtual interrupt lines.

- **Second Sender Role**: The third model evolved from the virtual interrupt model and is currently implemented in Fiasco. It adds a second sender role (preemption sender role) to each thread. In contrast to the virtual interrupt model, where there is one virtual interrupt for each thread that receives Preemption-IPC messages, this model adds a second sender role for each potential source of Preemption IPC. The kernel uses a thread's preemption sender role only for sending Preemption-IPC messages on behalf of that thread. The first sender role (IPC sender role) retains its original function of sending IPC and page-fault IPC messages. Having two sender roles for each thread allows the kernel to simultaneously engage in a Preemption IPC and a regular IPC for the thread. It is no longer necessary to suspend a regular IPC to be able to send a Preemption IPC.

As I already mentioned, the kernel does not allocate extra memory in order to queue preemption events. Instead it stores preemption events inside the scheduling contexts. When the expiration of a timeslice timeout or a deadline timeout requires the kernel to send a Preemption IPC, the kernel saves the type and timestamp of the preemption event in the thread's active scheduling context. That scheduling context is the one on which the owner thread either missed its deadline or on which the owner thread or another thread was preempted due to an expired time quantum. The kernel then passes a pointer to that scheduling context to the thread's preemption sender role. If the preemption sender role is currently not sending a Preemption IPC, it enqueues in the preempter's list of senders and attempts an IPC rendezvous with the preempter. Otherwise the submit function returns without taking any action. Once the preemption sender role has successfully sent a Preemption IPC to a preempter, it wipes the preemption event from the scheduling context and traverses the thread's list of scheduling contexts, trying to find another scheduling context with a preemption event. If no such scheduling context is found, all queued preemption events have been processed and the preemption sender role dequeues itself from the preempter's sender list.

Figure 3.12 shows the format of a Preemption-IPC message. Each Preemption IPC consist of two 32 bit words forming a 64 bit message, which is transferred in a register-only Short-IPC. The kernel sends the Preemption IPC on behalf of the preempted thread by using that thread's preemption sender role, thereby masquerading as the thread.

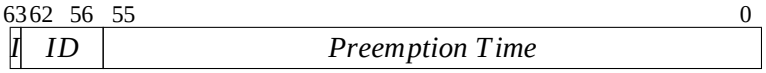


Figure 3.12: Preemption-IPC Message

The meaning of the fields is as follows:

- **Preemption Time**: The preemption time corresponds to the 56 least significant bits of the kernel clock at the time the preemption event occurred. Because the kernel clock counts in microseconds, the timestamp is wide enough to count 2284 years without wrapping around.
- **ID**: The 7 bit ID field corresponds to the identification number of the scheduling context on which the preemption event occurred. It is wide enough to describe up to 127 reservation scheduling contexts per thread.
- **I**: The single bit type identifier describes the type of preemption event that occurred.
  - 0 : Deadline Miss
  - 1 : Timeslice Overrun

Note that there is still a reduced chance that preemption events will be lost, namely if a thread is preempted again on a scheduling context that already had a pending preemption event for which the preemption sender role had not yet sent a Preemption IPC. This is because each scheduling context can queue only one preemption event, so that later events overwrite earlier events on the same scheduling context. If unlimited queueing of preemption events is desired, the kernel has to allocate memory to store all unsent preemption events, which would allow a malicious thread to perform a denial-of-service attack against the kernel, by permanently generating preemption events without its preempter ever becoming ready to receive the corresponding Preemption IPC.

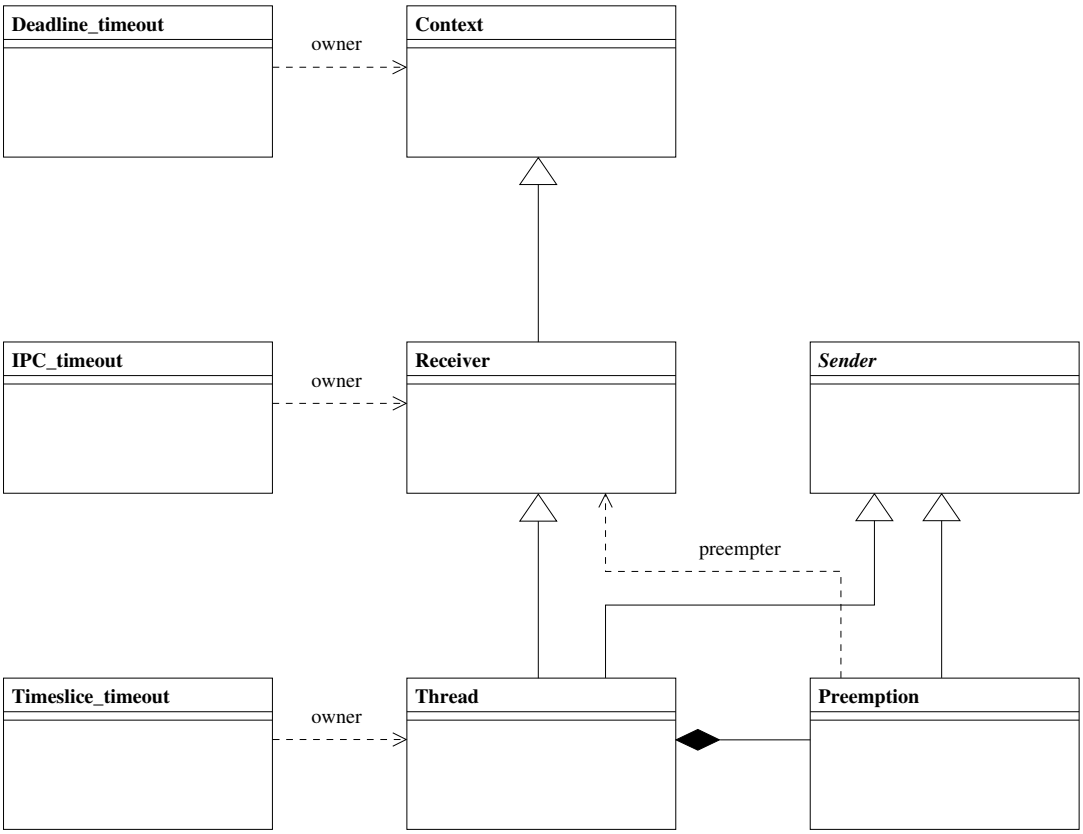


Figure 3.13: Preemption-IPC Related Classes

Figure 3.13 shows a class diagram with classes related to Preemption IPC. As in older implementations of the Fiasco microkernel, the `Thread` class inherits from the `Sender` and `Receiver` classes. Whereas the `Sender` class is a pure role class, the `Receiver` class inherits from the `Context` class, which implements execution contexts [9]. In addition to a thread's first sender role, which the thread contains because it derives from `Sender`, I added a new `Preemption` class, which also derives from `Sender` and which each `Thread` aggregates. Each `Preemption` sender role knows a `Receiver`, the thread's preempter, which receives the Preemption-IPC messages sent by that sender role.

On the left side of the figure I present the three different timeout classes and their relation to other classes. The owner of a `Deadline_timeout` is a `Context`, the owner of an `IPC_timeout` is a `Receiver` and the owner of the `Timeslice_timeout` is a `Thread`. The Fiasco design methodology mandates that circular dependencies between classes must be avoided to facilitate a hierarchy of different classes and subsystems. Sometimes it requires bizarre constructions to accomplish this goal. I will give two examples:

- Because each thread aggregates a deadline timeout in its TCB, there is a dependency from `Thread` to `Deadline_timeout`. In order to be able to send a "Deadline Miss" Preemption IPC, the deadline timeout must know the thread's preemption sender role. Therefore we have another dependency from `Deadline_timeout` to `Thread`. The current implementation avoids this circular dependency by passing a pointer to the thread's preemption sender role when

initializing the thread's deadline timeout. We then have a dependency from `Deadline_timeout` to `Preemption`, which is no longer circular.

- When the kernel selects a scheduling context as the current scheduling context, it programs a timeslice timeout for the point in time when the scheduling context's time quantum is exhausted. Therefore we have a dependency from `Context` to `Timeslice_timeout`. When a timeslice timeout expires and the kernel wants to send a "Timeslice Overrun" Preemption IPC, the timeslice timeout must know the thread's preemption sender role, creating a dependency from `Timeslice_timeout` to `Thread`. Due to inheritance `Thread` depends on `Context`. This circular dependency cannot be avoided in a fashion similar to the previous example, because the timeslice timeout does not belong to a particular thread and its owner changes as the current scheduling context changes. To break this circular dependency the scheduler code must not work with timeslice timeouts, but must instead use a generic timeout pointer, which is a pointer to the timeslice timeout in disguise. We then have a dependency from `Context` to `Timeout` instead.

3.9 Time Donation

The Dresden Realtime Operating System (DROPS), which uses the Fiasco microkernel, implements a multiserver environment (L4Env) [14] on top of the microkernel. Client threads use the system services through IPC calls to different servers. Each server provides a specific service to its clients; for example there are semaphore servers, name servers, file providers and memory managers.

For the calculation of reservation times and priorities, the quality-assuring-scheduling model assumes that tasks to be admitted are independent of each other. In our microkernel-based system, where the vast majority of communication between clients and servers uses IPC[3], this assumption is often not true. The different servers of the environment can be modelled as resources, and different threads using the provided services contend for these resources. In combination with the different priorities of the client threads, this leads to *priority inversion* problems as shown in Figure 3.14.

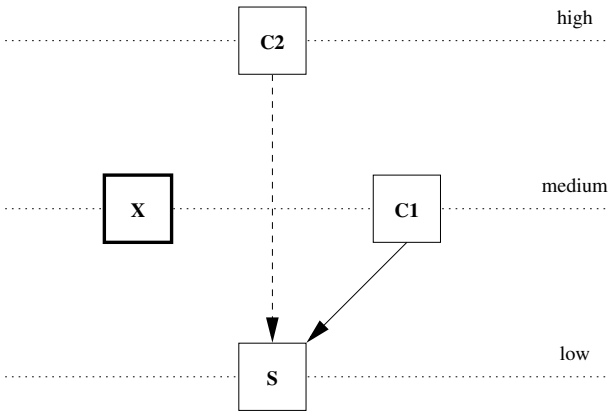


Figure 3.14: Priority Inversion

[3]Inter Process Communication

In this example there is a medium-priority client thread C_1 using the services of the low-priority server thread S. C_1 has sent a request to S and is now blocked, waiting for S to reply. While S is handling C_1's request, another thread X becomes ready and preempts S because of its higher priority. Shortly thereafter another client, C_2, starts running and preempts X. When C_2 wants to use the services of S it cannot, because the server is currently busy handling the request of C_1. C_2 therefore blocks, trying to send its request to S, while X continues to run. Neither S nor C_1 or C_2 are making any progress, even though C_2 has a priority higher than X. Priority inversion can be avoided if C_2 donates its high-priority scheduling context to S until S has replied to C_1. The achieved effect of the time donation is that S inherits C_2's high priority; therefore downward donation implements the *priority inheritance* algorithm [24].

Another issue that needs to be addressed is *time accounting*, when client threads use the services provided by other server threads. In Figure 3.14 the client thread C_1 uses the services of S. The admission server must either assign S enough time to handle client requests using S's own time, or client threads must provide time resources to S for the time during which S services their request. The time for an IPC call must be accounted either completely to the client or completely to the server.

Because *IPC performance* is a critical factor for the success of microkernel-based systems, L4 developers have put much effort into making IPC as fast as possible. One performance optimization is to directly switch to the receiver of an IPC after sending the message, then blocking, waiting for the reply. Directly switching to the receiver bypasses the scheduler and avoids the overhead of choosing the next thread to run. I discuss this overhead in Section 5.2.

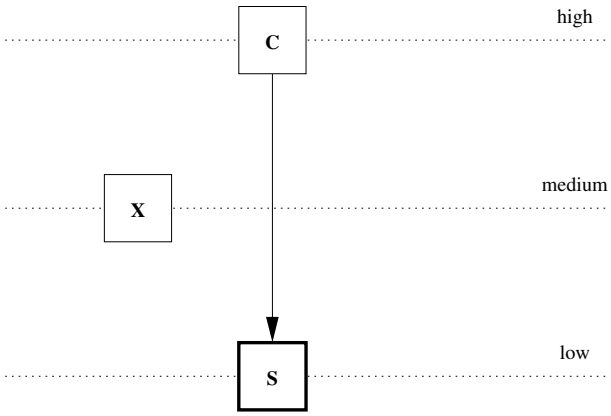


Figure 3.15: Priority Scheme Violation

The direct switch is not problematic, if the receiver has a priority higher than or equal to that of the sender, because in this case there are no other ready threads with a priority higher than that of the sender. If this were not the case, the sender would not be the current thread. However, if the receiver has a lower priority than the sender, then directly switching to the receiver may violate the priorities in the system as shown in Figure 3.15.

In this example the high-priority client thread C has sent its request to the low-priority server S. C now blocks, waiting for the reply, and S is now ready. To avoid calling the scheduler, C directly switches to S, ignoring the fact that thread X is also ready. This direct switch is a violation of the priority scheme, because X has a higher priority than S and should run instead. This violation can be avoided using priority inheritance. Since S has a lower priority, C can donate its high-priority

scheduling context to S. S's priority is then effectively boosted to that of C, which was higher than that of thread X.

In order to maintain correct scheduling behaviour, it is necessary for the kernel to provide a mechanism that allows threads to donate the current scheduling context with its associated time quantum and priority to another thread when engaging in an IPC operation, to:

1. avoid priority inversion by using priority inheritance,
2. provide a mechanism that supports time accounting across IPCs, and,
3. retain maximum IPC performance by switching directly to the receiver of an IPC message, even if the receiver has a lower priority than the sender, while still honoring the priorities of other threads.

The current L4 IPC implementation tries to avoid priority violations by always donating the current scheduling context to the receiver of an IPC. The kernel accomplishes this by switching to the receiver using `switch_to` without also switching to the receiver's active scheduling context[4]. Nevertheless there is a problem with this approach: the sender only donates one current scheduling context to the receiver. Once the time quantum on that current scheduling context is depleted or the scheduler selects a new current scheduling context, the donation ends.

Consider the example in Figure 3.15. If C performs an IPC call to S, it sends the request and afterwards blocks until the reply arrives. During the IPC C donates the current scheduling context to S. While S computes the reply, an asynchronous event, such as the release of another thread due to an interrupt or the depletion of the time quantum on the current scheduling context, can cause the kernel to invoke the scheduler. Because C is now blocked, the scheduler will select X as next thread to run. Ideally the scheduler should recognize that C has donated its scheduling context to S and should activate C's scheduling context as current scheduling context and S as current thread. However, this requires the scheduler to be able to recognize donation dependencies between threads engaged in an IPC.

Let us now explore how such a dependency could be specified during an L4 IPC call to accomplish various donation semantics. Figure 3.16 illustrates the message transfer phases during an IPC call. A client thread C sends a request to a server thread S. The server then performs some work on behalf of the client and sends back a reply. The client uses an L4 IPC *call*, which consists of a send phase for the request and a receive phase for the reply. The server uses the L4 *reply-and-wait* IPC, which is similar to call, except that the server receives the request using the receive phase of the previous call, sends back the reply using the send phase of the current call, and then uses the receive phase of the current call to wait for the next request from any thread.

During Long-IPC, page-faults can occur in the sender's or the receiver's address space. Such a page-fault blocks both the sender and the receiver until the faulting thread's pager supplies a memory mapping to resolve the fault. Therefore donation applies not only to regular IPC, but to page-fault IPC as well.

Senders of an IPC can set one of the following donation hints to specify the donation dependency as desired by the thread. The kernel evaluates the hint during the IPC rendezvous.

[4]There is an undocumented feature involving the deceiting bit which prevents the automated switch and invokes the scheduler.

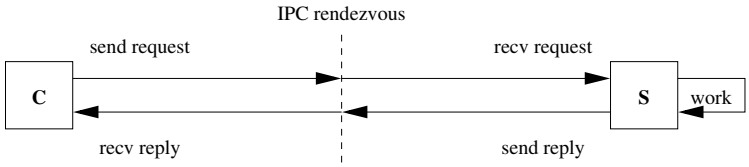


Figure 3.16: IPC Call

1. **Classic L4 IPC**:
   When a thread chooses this option, the kernel switches to the thread's IPC partner without switching to the partner's active scheduling context, thus donating the current scheduling context to the partner. However, the kernel does not create a dependency between both parties. If a page-fault occurs in the thread's or the partner's address space, the kernel switches to the faulting thread's pager, so that donation automatically extends to a nested page-fault IPC in either address space. This behaviour is already implemented in the Fiasco microkernel. Despite the fact that the duration of the donation is unpredictable, because the next invocation of the scheduler ends the donation, this behaviour must be preserved to ensure backward compatibility.

2. **Donating Receive**:
   Here the thread performing an IPC donates the current scheduling context to the IPC partner and then blocks, trying to receive an IPC message from the partner. The kernel must create a donation dependency from the thread to the partner and maintain it until the receive phase has finished. If a page-fault occurs in either address space during message transfer, the kernel temporarily replaces the dependency with a new dependency from the thread to the pager that handles the fault. Once the fault has been resolved, the kernel restores the old dependency.

3. **Donating Send**:
   In this case the thread performing an IPC donates the current scheduling context to its IPC partner only until the partner has received the message. The kernel must create a donation dependency from the thread to its partner and remove the dependency once the message transfer has finished. If a page-fault occurs in either address space during the message transfer, the kernel temporarily replaces the dependency with a new dependency from the thread to the pager that handles the fault. Once the fault has been resolved, the kernel restores the old dependency. This option is useful for servers using *reply-and-wait*, because it allows them to donate time to the client while sending back the reply, without allowing the client to use the donated time for its next request.

4. **Donating Call**:
   The donating call is a combination of a donating send and a donating receive. The thread performing a donating call donates the current scheduling context to the IPC partner during the entire IPC – the message transfer of the request to the partner, the partner's work phase and the message transfer of the reply from the partner back to the thread. The kernel must create a dependency from the thread to its partner and maintain it until the entire IPC has finished. This option is similar to the first option, except that the duration of the donation is deterministic.

5. **No Donation**:
   The thread performing an IPC does not donate the current scheduling context; therefore the kernel does not create a donation dependency.

Receivers of an IPC can set two different donation hints:

1. **Accept/Reject Donation**:
   Using this hint, a receiving thread can specify whether it wants to accept or reject time donations from sending threads. If the sender wants to perform an IPC with donation and the receiver rejects it, no donation occurs.

2. **Donate During Page-Faults**:
   Because the receiver of an IPC also blocks if a page-fault occurs during the message transfer, the receiver can specify whether it also wants to donate its time to the pager that handles the fault. In that case the kernel adds a dependency from the receiver to the pager. If both the sender and the receiver of an IPC donate time to the pager during a page-fault, the donated time is provided by the thread with the higher priority.

A donation dependency from one thread to another can only exist if the first thread is engaged in an IPC with the second thread. If a thread invokes an IPC system call, the kernel evaluates the donation hint and sets donation flags for the send and receive phase in the thread's status word accordingly. When the send or receive phase finishes, the kernel also clears the donation flag for the phase in the status word.

The scheduler can then traverse donation dependencies between threads according to the following algorithm:

1. By checking the `Thread_ready` flag in the thread's status word, the scheduler determines if the thread is ready to run. If it is, the scheduler dispatches it and the algorithm stops.

2. Otherwise the kernel checks whether the send or receive donation flag is set in the thread's status word. If neither is set, then no donation dependency exists. Because the thread is not ready, the scheduler removes it from the ready list and the algorithm stops.

3. Otherwise a dependency exists. The donation target can be found in the thread's IPC partner field. The scheduler then repeats the algorithm for the donation target.

3.9.1 Downward Donation

Let us now examine how the scheduler selects a new current scheduling context and a new current thread to run on that scheduling context in the presence of donation. For now we assume that each thread only donates time to lower-priority threads. I will remove this limitation in Section 3.9.2. This downward-donation scheme implements the *priority inheritance* protocol [24]. Figure 3.17 shows two examples. On the left side there is a ready list with three threads: a high-priority thread *A*, a medium-priority thread *B* and a low-priority thread *C*. *A* has performed a donating IPC call to *B*, while *B* has to engage in a donating IPC call with *C* before it can reply to *A*. When the scheduler traverses the ready list, it finds that *A* is the thread with the highest priority currently enqueued. Executing the donation dependency tracking algorithm, the scheduler recognizes that *A* is blocked in a donating IPC to *B* and that *B* is blocked in a donating IPC to *C*. Because *C* is ready to run, the scheduler selects *A*'s active scheduling context as current scheduling context and *C* as current thread. As a result *C* now executes on *A*'s priority and the execution time of *C* is accounted to *A*.

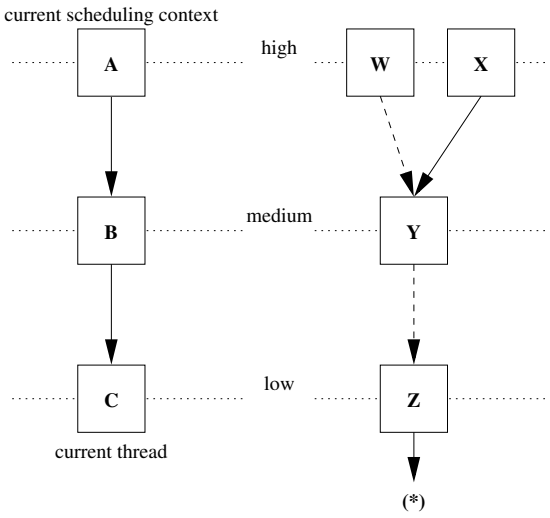


Figure 3.17: Downward Donation

The situation becomes more complicated if the scheduler finds a dead end while traversing the dependency chain. Figure 3.17 shows such a scenario on the right side. The ready list contains four threads *W*, *X*, *Y* and *Z*. *X* is currently engaged in a donating IPC with *Y*. *W* also attempts a donating IPC to *Y* but blocks, because *Y* is busy handling *X*'s request. Before *Y* can reply to *X* it has to perform a donating IPC to *Z*. However, *Z* is currently sleeping. In this case the scheduler cannot activate the thread at the end of the donation chain, because *Z* resembles a dead end. Possible dead ends are:

- threads performing a *sleep* IPC.
- threads performing an *open-wait* IPC.
- threads waiting for an interrupt.
- threads waiting for a Preemption IPC.

In such a situation multiple donation chains can be blocked on a dead end and none of the threads in any such chain can execute. The scheduler can be implemented to either skip dead chains or remove dead chains from the ready list, similar to the way it removes threads that are not ready. I will briefly discuss the advantages and disadvantages of both options.

Removing Dead Donation Chains

Removing dead chains from the ready list has the advantage that the ready list only contains threads that are either ready or part of a donation chain with no dead end. Scheduling decisions can be made quickly; the scheduler selects the active scheduling context of the thread with the highest priority as current scheduling context and follows the donation chain from that thread. At the end of that chain it finds the thread that should become the current thread.

However, removing dead chains has a serious drawback. Consider what happens in Figure 3.17 if the donation dependency between *Y* and *Z* breaks because the IPC send-timeout for *Y* is triggered. Suddenly the whole donation tree that can be spanned, using the thread at the broken dependency link as root node, becomes ready again and must be reenqueued. In our example the donation tree consists of threads *Y*, *W* and *X*. The leaf nodes of the tree (*W*, *X*) are threads providing the donated time and their active scheduling context is a candidate for becoming the current scheduling context. The root node of the tree (*Y*) is the end of all donation chains and will become the current thread and receive the donated scheduling context from either *W* or *X*. Because the scheduler needs both the root node and all leaf nodes, the whole donation tree must be reenqueued atomically. If the tree consists of hundred or thousand threads, the kernel suffers from unbounded interrupt latency.

Skipping Dead Donation Chains

Alternatively the scheduler could skip over a donation chain after determining that it leads to a dead end. This results in heavily degraded scheduler performance, because the scheduler must potentially traverse many threads in dead donation chains before finding a thread that is ready or not part of a chain leading to a dead end. Traversing many donation chains is likely to cause many TLB and cache misses. Nevertheless, this option seems to be preferable over the previous solution, because the traversal of donation chains can be preemptible. However, if the scheduler is preempted while inspecting the ready list and traversing the donation chains, then an interrupt handler can wake up another thread and enqueue it in the ready list. In such a situation the scheduler must start again from the beginning, otherwise it risks stepping over newly enqueued threads.

3.9.2 Upward Donation

In this section I eliminate the limitation of only allowing donation to lower-priority threads and extend the donation mechanism to allow upward donation as well, to be able to build systems that implement the *stack-based priority ceiling* protocol [24]. The protocol assigns each resource a priority that is higher than the priority of any thread that ever requests the resource. When such a resource is allocated to a thread, the thread's priority is boosted to the priority of the resource.

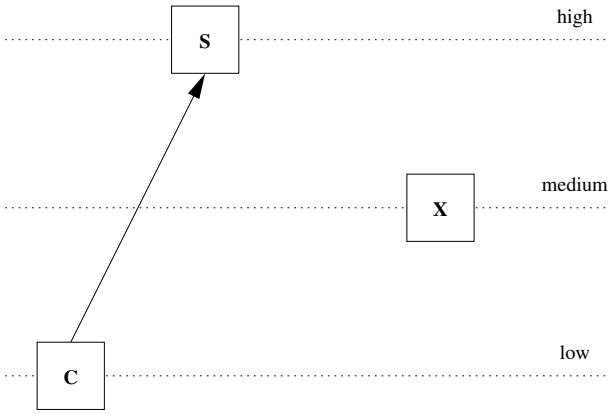


Figure 3.18: Upward Donation

Figure 3.18 shows a high-priority server *S* and a low-priority client *C*. When *C* performs a donating IPC to *S*, then *S* consumes the time on the scheduling context of *C*. However, *S* must retain its high priority. If the kernel allowed *C* to drag down *S* to the low priority, then the system would suffer from priority inversion as soon as *X* becomes ready to run. Theoretically *S* could run on *C*'s low priority as long as *X* is blocked, but *S* must use its own high priority when *X* becomes ready.

Because the stack-based priority ceiling protocol demands that *S* consumes *C*'s time, but retains its own high priority, the kernel cannot simply let *S* run on *C*'s scheduling context, because that would lower the priority of *S* to that of *C*. Instead the kernel must temporarily elevate the priority on *C*'s scheduling context to that of *S*. To distinguish the elevated priority from the original priority of the scheduling context, I call it *boost priority*. The kernel then enqueues threads in the ready list according to the boost priority of their active scheduling context instead of using the original priority. This requires that the kernel update the boost priority on a scheduling context whenever the scheduling context is donated to another thread, as shown in Figure 3.19.

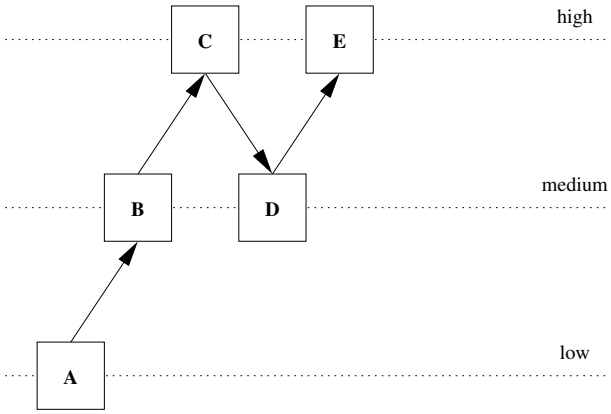


Figure 3.19: Boost-Priority Calculation

In this example thread *A* has donated its low-priority scheduling context to the medium-priority thread *B*, thus boosting the priority on *A*'s scheduling context from low to medium. *B* then further donates *A*'s scheduling context to *C*, thereby boosting the priority on *A*'s scheduling context from medium to high. *C* then donates the scheduling context to *D* and *D* donates it to *E* using IPC calls. During the establishment of the latter two donation dependencies, the boost priority on *A*'s scheduling context does not change.

Consider what happens when the donation dependency between *D* and *E* is torn down, because the IPC between *D* and *E* finishes. The kernel must recalculate the boost priority of *A*'s scheduling context, because *E*'s priority is equal to *A*'s boost priority and *E* is no longer part of the donation chain. This could indicate that the boost priority on *A*'s scheduling context must now be lowered to medium, unless another thread in the donation chain also has a priority equal to *A*'s boost priority – thread *C* in this example.

Unfortunately the recalculation requires that the scheduler traverse the donation chain starting from the owner of the scheduling context. In this example the kernel has to track the donation dependencies starting from *A*, then finding *B* and finally *C*, at which point it can stop, because it knows that *C* can keep the boost priority on *A*'s scheduling context at the high level. Note that such a recalculation is necessary whenever a dependency created during an IPC with donation is torn down, for example

when an IPC with donation finishes or times out. The recalculation of the boost priority heavily degrades IPC performance.

I have evaluated various possibilities to recalculate the boost priority on a scheduling context lazily. The boost priority only needs to be up-to-date when the owner of the scheduling context is going to be enqueued in the ready list according to the boost priority, or when another thread is woken up and the kernel must decide whether that thread should preempt the current thread. Such a scenario occurs when a hardware interrupt wakes up an IRQ thread attached to the interrupt. The kernel then has to recalculate the boost priority of the current thread before it can decide whether the interrupt thread has a higher priority. The lazy recalculation of the boost priority is not a viable solution because it deteriorates interrupt latency.

3.10 Locks and Helping

The Fiasco microkernel implements a form of wait-free locking as described in [10] to synchronize kernel objects. Each lock knows the thread which holds the lock. Additionally each lock implements a helper stack. If a thread *H* wants to acquire a lock that is currently held by another thread *O*, then *O* is per definition ready[5]. *H* must have a priority that is equal to or greater than that of *O*, otherwise it would not run instead of *O*. Because *H* cannot acquire the lock immediately, it puts itself on top of the lock's helper stack and helps *O* to release the lock, by donating its scheduling context, and thus its time and priority, to *O*. A thread in the helper stack remains ready and continues to donate to the lock owner each time the scheduler reactivates it, until the lock owner releases the lock and passes it to the thread on top of the helper stack. Because the helping mechanism is a low-level form of donation, I will use the term *helping* to distinguish it from the high-level IPC donation mechanism.

I identified two situations where the current helping implementation does not work as intended and passes the lock to the wrong thread:

1. **Multiple helper threads with equal priority**:
   If there are multiple threads with equal priority trying to acquire a lock that is currently held by another thread, then each helper puts itself on top of the helper stack. Although the helper stack is sorted by ascending priority, the order of helper threads with equal priority in the stack depends only on the order in which they tried to obtain the lock. When the time quantum on the scheduling context provided by the thread on top of the helper stack is depleted, the round-robin scheduler will select the active scheduling context of another helper thread with equal priority as new current scheduling context. If the lock owner then releases the lock, it passes the lock to the thread on top of the helper stack, instead of passing it to the helper thread which provided the time to help release the lock. This can lead to starvation of threads in the helper stack.

2. **Priority change of helper threads**:
   If the priority of a thread, which is enqueued in the helper stack of a lock, changes, the kernel does not resort the helper stack according to the changed priorities. Such a scenario happens when the thread on top of the helper stack passes its scheduling context to the lock owner due to helping and the lock owner exhausts the time quantum on that scheduling context, thereby activating the helper thread's next scheduling context, which typically has a lower priority. When the lock owner finally releases the lock, it passes the lock to the helper on top of the

[5] A thread holding a lock must not sleep.

helper stack, but that thread may no longer be the helper thread with the highest priority. The resulting effect is priority inversion.

I therefore propose to change the lock release mechanism in such a way that the kernel always activates the helper with the highest priority – which is per definition the thread that provided the current scheduling context to the lock owner to help it release the lock.

Figure 3.20 shows a complex scenario. Thread O is the low-priority lock owner. Enqueued in the lock's helper stack are the helper threads H_1 and H_2 with high priority and the helper thread H_3 with medium priority, all of which want to acquire the lock. Thread D wants to delete O and has therefore locked O, but before it can proceed, O must first release all locks it holds. Finally, there are three high-priority threads T_1, T_2 and T_3, which do not wish to acquire the lock. T_1 donates its time directly to O, for example by means of a donating IPC call, T_2 donates its time to D and T_3 donates its time to H_1.

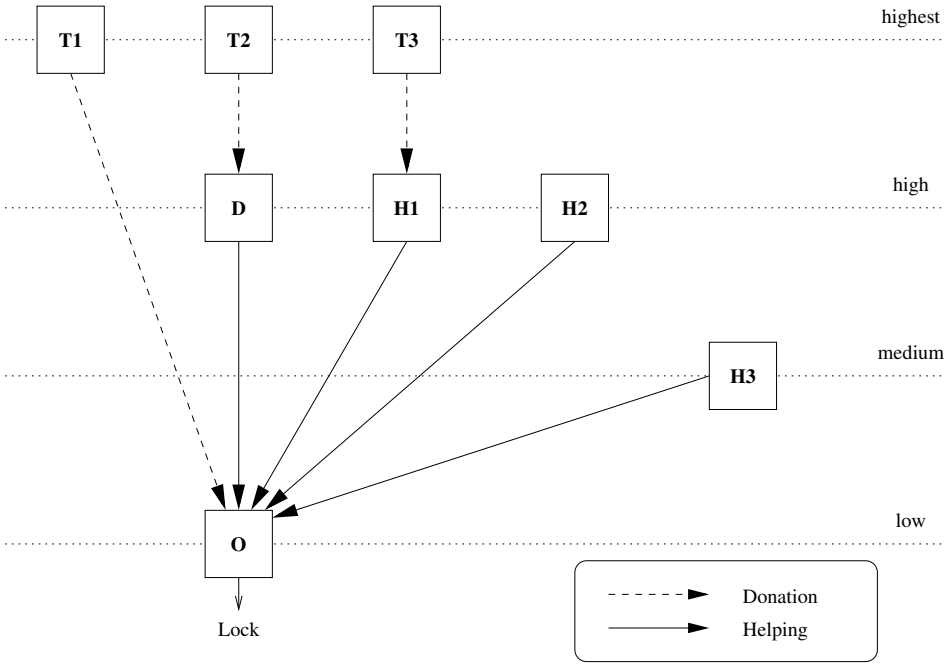


Figure 3.20: Helping Mechanism

When O releases the lock, the kernel must take one of the following actions, depending on which thread (T_1, T_2 or T_3) provided the current scheduling context to help release the lock:

- T_1 donated its scheduling context directly to O and no helping is involved; therefore, after releasing the lock, O should continue to run. However, in this scenario O is locked and shall be deleted, so O only runs if it holds more locks that it has to release. Otherwise the kernel must invoke the scheduler to select a new thread to run. The lock that O just released remains unlocked, despite the fact that there are other threads wanting to acquire the lock. The priority of the helper threads is currently not high enough to run instantly.
- T_2 donated its scheduling context to thread D, which wants to delete O. Because O holds a lock, D passes T_2's scheduling context to O to help it release the lock. Once O has released

the lock, D must continue running on T_2's scheduling context. The lock that O just released remains unlocked and D can proceed to delete O.

- T_3 donated its scheduling context to the helper thread H_1, which then passes it on to O. Even if H_2 is top of the helper stack, once O releases the lock, H_1 continues to run on T_3's scheduling context. H_1 can then acquire the lock that O previously held and become new lock owner.

To switch to the thread which provided the current scheduling context to help release the lock, the lock owner could invoke the scheduler upon releasing the lock, which would cause the scheduler to reselect the same current scheduling context. By traversing the donation and helping dependencies the scheduler would then find the correct thread that should run next. Unfortunately this solution adds a scheduler invocation to each lock release operation. Each IPC comprises at least two lock release operations, because the sender locks down the receiver during both message transfers.

Based on the observation that when the lock owner releases the lock, the current scheduling context does not need changing, I developed a solution that will activate the correct next thread with a single `switch_to`, except when the lock owner is to be deleted and releases the last lock it holds. To track the most recent helper of a thread holding a lock, my solution adds a `lock_helper` field to each thread control block. When a thread switches to another thread using `switch_to`, thereby donating the current scheduling context to the destination thread, it specifies whether it is helping the destination thread or not. If a thread is helping the destination thread, it sets the destination thread's `lock_helper` field to the helping thread, otherwise it sets the `lock_helper` field to the destination thread.

In Figure 3.20 the threads D, H_1, H_2 and H_3 are helping the lock owner O. When T_1 is active, it switches to O, but no helping is involved; therefore T_1 sets O's `lock_helper` field to O. When T_2 becomes active later, it switches to D and because there is no helping, T_2 sets D's `lock_helper` field to D. If D then switches to O to help O release its locks, it overwrites O's `lock_helper` field with D. It follows that the thread which most recently donated the current scheduling context to O sets O's `lock_helper` field to either itself or to O. When O later releases the lock, the kernel only has to inspect O's `lock_helper` field to decide which thread should run next. If the `lock_helper` field contains O, then the current scheduling context was not provided by a helper and O continues to run, unless O is locked itself and holds no more locks. However, if the `lock_helper` fields contains a different thread, the kernel knows that it can directly switch to the thread, because that thread provided the current scheduling context via helping and should therefore run next. As a side effect of this solution the stack of helper threads associated with each lock becomes obsolete and can be removed.

The code for the lock release changes as follows:

```
if (_lock_owner->helper() != _lock_owner)
  switch_to (_lock_owner->helper());

if (_lock_owner->lock_cnt() == 0 && _lock_owner->donatee())
  schedule();
```

The first statement checks whether the `lock_helper` field is different from the lock owner. In that case the kernel directly switches to the most recent helper thread. The second statement checks if the lock owner released its last lock and whether the lock owner is locked itself. In that case the kernel must invoke the scheduler to determine the next thread to run, because the lock owner is to be deleted and must not continue to run.

Chapter 4

Implementation

4.1 Scheduling Modes

With the addition of periodic threads to Fiasco, the microkernel now supports two different scheduling modes:

- **Conventional Mode**
  When a thread executes in the default conventional mode, the kernel never activates any of the reservation scheduling contexts the thread may have. Whenever the thread executes, it runs on its own regular scheduling context or on a donated scheduling context.

- **Periodic Mode**
  A thread executing in periodic mode is a periodic thread. Each time a new period begins for the thread, the kernel reactivates the threads's first reservation scheduling context, if it has one. During periodic execution the kernel checks for deadline misses and timeslice overruns of the thread and reports such events to the thread's preempter using the Preemption-IPC feedback mechanism.

When the kernel creates a new thread, the thread begins executing in conventional mode. If the new thread wants to enter periodic mode, its admission server must first configure the thread's periodic parameters (scheduling contexts, period length) in the kernel and then invoke an *rt_begin_periodic* call, telling the kernel when the destination thread should begin with its periodic execution. The thread itself must signal the kernel its intention to enter periodic mode by performing an *rt_next_period* call before the point in time when periodic execution shall begin. This scheme ensures that a malicious thread cannot disrupt other periodic threads by configuring erroneous scheduling parameters, and also ensures that a malicious admission server cannot force a thread to transition to periodic mode without the thread's consent. When the admission server wants a thread to leave periodic mode, it executes an *rt_end_periodic* call and the destination thread immediately returns to conventional mode. For this operation the thread's consent is not required. When a thread accepts an admission for periodic mode, it automatically agrees that the admission server can revoke the admission at any time. In such a situation the thread can renegotiate for a new admission.

Figure 4.1 shows a state transition chart for the different scheduling modes. When an admission server performs an *rt_begin_periodic* call, the kernel programs a deadline timeout for the destination thread

for the point in time when periodic execution shall begin. When the thread executes its *rt_next_period* call, the kernel sets the `Thread_delayed_deadline` bit in the thread's status word, which indicates that the thread is waiting for the beginning of its next period.[1] If this bit is set, the thread blocks inside the kernel until the bit is cleared at the expiration of the thread's deadline timeout. Calls to *rt_begin_periodic*, *rt_end_periodic* or *rt_next_period*, which are invalid in a particular state, are indicated with a dashed line and return an error to the caller.

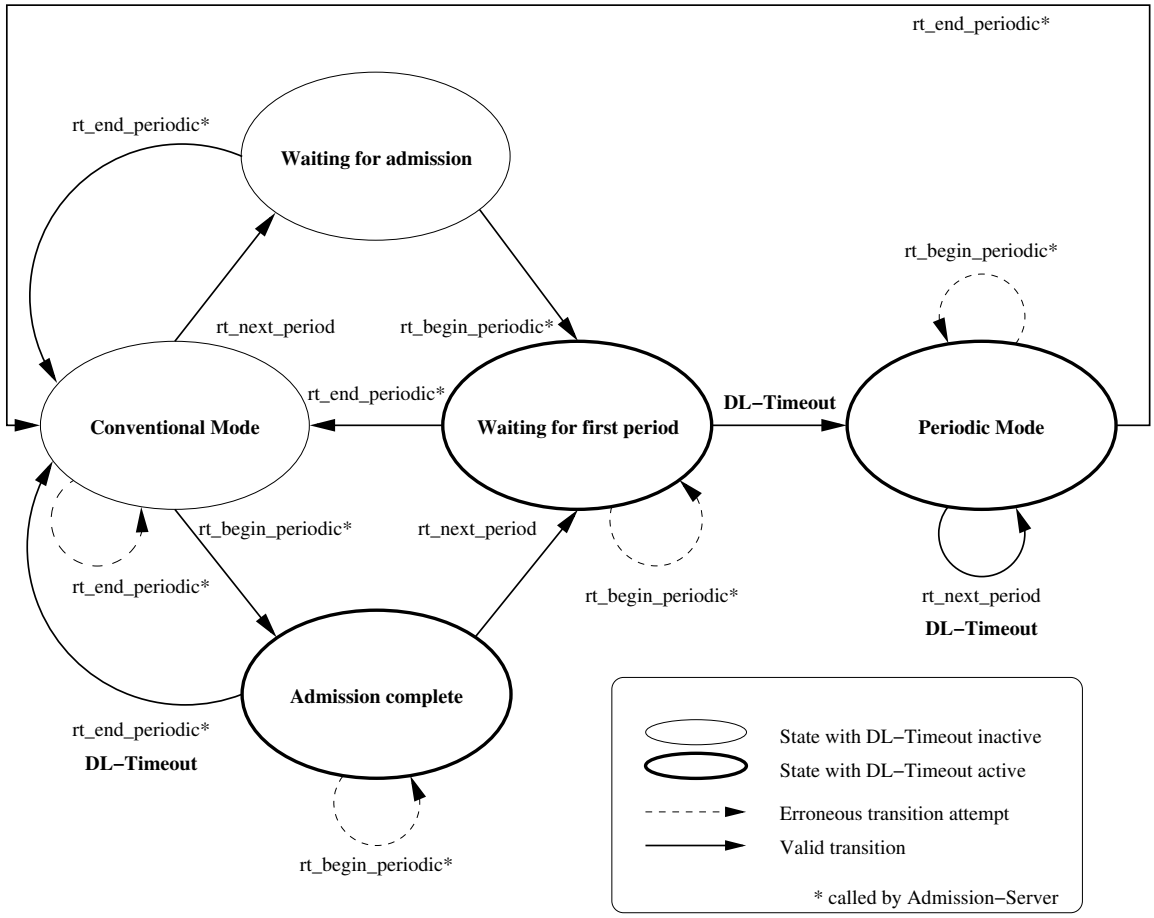


Figure 4.1: Scheduling-Mode State Transition Chart

Initially the thread is in the state "Conventional mode". If the thread first performs its *rt_next_period* call, it enters the state "Waiting for admission". In this state the thread is blocked inside the kernel because its `Thread_delayed_deadline` bit is set. Once in this state, the thread waits for admission to complete. If the admission server admits the thread, it executes an *rt_begin_periodic* call, which causes the kernel to program the thread's deadline timeout to the beginning of the thread's first period, and the thread enters the "Waiting for first period" state. If the admission server rejects the thread, it executes an *rt_end_periodic* call, which causes the kernel to clear the `Thread_delayed_deadline` bit and wake up the thread. The thread then continues execution in conventional mode. If the thread is in state "Conventional mode" and the admission server performs its *rt_begin_periodic* call before the thread performs its *rt_next_period* call, then the thread enters the "Admission complete" state, where

[1] Here the next period is the thread's first period

the thread's deadline timeout is already programmed to the beginning of the thread's first period. If the thread executes its *rt_next_period* call in a timely manner, that is, before the deadline timeout expires, it enters the state "Waiting for first period". If the thread is late and the timeout expires before the thread performs its *rt_next_period* call, then the thread remains in conventional mode. Once the thread is in the "Waiting for first period" state, the `Thread_delayed_deadline` bit is set in the thread's status word and the thread's deadline timeout is programmed. When the timeout expires, the transition succeeds, the thread enters the state "Periodic mode" and begins with periodic execution in its first period. If the admission server performs an *rt_end_periodic* call before the timeout expires, the kernel cancels the deadline timeout, clears the `Thread_delayed_deadline` bit and the thread remains in conventional mode.

Figure 4.2 summarizes the kernel's internal representation of the different states. When executing in periodic mode, the kernel distinguishes between strictly periodic threads and periodic threads with minimal interrelease times using a second bit in addition to the "Mode Bit".

Mode	Deadline-Timeout set	Delayed-Deadline Bit	Mode Bit
Conventional mode	–	0	0
Waiting for admission	–	1	0
Admission complete	√	0	0
Waiting for first period	√	1	0
Periodic mode	irrelevant	irrelevant	1

Figure 4.2: Scheduling-Mode Internal Representation

4.2 System-Call Extensions

The quality-assuring scheduling model is not part of any current L4 API specification. For the implementation of my design I therefore had to make modifications to an existing L4 API. The Fiasco microkernel currently supports the stable version 2 (V.2) and the experimental version 0 (X.0) API. Work is in progress to add the latest experimental version 2 (X.2) API to Fiasco, which will eventually evolve into the next stable version 4 (V.4). Because the X.2 support in Fiasco was still incomplete at the beginning of my work, I decided to modify the V.2 API, which is widely used and well understood. However, the Fiasco port to the ARM architecture [25] only supports the X.0 API, so I later expanded the scope and designed all API modifications such that they are compatible with both V.2 and X.0. I put a strong focus on backward compatibility with existing APIs, so that my work does not require any fundamental changes to L4, but rather extends existing L4 functionality with new features.

Instead of adding new system calls for additional features, I decided to extend the following three L4 system calls with additional functionality:

- **thread_schedule**
  L4 uses this system call for the configuration of regular scheduling parameters of a thread. Therefore I implemented all QAS functions dealing with the configuration of reservation scheduling contexts on top of it. These include:
  - Adding and removing reservation scheduling contexts (*rt_add*, *rt_remove*)

  - Setting a thread's period length (*rt_period*)
  - Beginning periodic execution (*rt_begin_periodic*)
  - Stopping periodic execution and returning to conventional mode (*rt_end_periodic*)

- **thread_switch**
  Using this system call, a thread can yield the current scheduling context or donate it to another thread. I therefore considered it the optimal place to add the function to voluntarily release reservation scheduling contexts (*rt_next_reservation*).

- **ipc**
  When a thread wants to wait for the beginning of the next period, it typically sleeps until the next period begins or additionally waits for a specific event, such as the reception of an IPC message. Both sleeping and message transfer are implemented in L4 using the IPC system call, so I extended it with the functionality to wait for the beginning of the next period (*rt_next_period*).

The ABIs for the V.2 and X.0 API define which processor registers are used to pass system-call parameters to the kernel and which registers contain the return parameters of the system call. Transferring parameters in registers is fast, because the kernel does not need to access memory; therefore, performance does not suffer from additional cache or TLB misses. For this reason the L4 ABIs try to pass as much information as possible in the general purpose processor registers, leaving little to no register space for extensions. The L4 system call with the most excessive register use is the *ipc* system call, used for message transfer and for establishing memory mappings. As specified in the L4 reference manual [19], it uses all seven available general purpose registers of the x86 architecture, so that I had to redefine some bits in one of the registers so as to avoid the need of passing parameters on the stack. This is because many of the extensions introduced during my work, such as absolute timeouts and Preemption IPC, are based on or can be used together with the *ipc* system call. I chose to redefine the bits in the destination thread ID, which contain the *chief* number. The clans and chiefs concept [17] was never completely implemented in the Fiasco microkernel and is currently only used for task creation and deletion rights. I could not use the *site* or *nest* bits of the thread ID, because they do not exist in the X.0 API.

Figures 4.3 and 4.4 show the format of a thread ID for the V.2 and X.0 APIs with the redefined bits in the *chief* field.

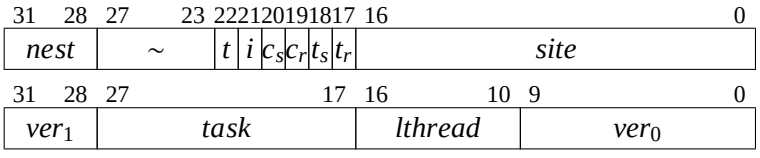


Figure 4.3: Thread ID (V.2)

In the V.2 API the chief field is located in bits 17 through 27 of the high word of the thread ID. In the X.0 API the chief field is located in bits 24 through 31 of the thread ID[2]. The current implementation redefines six of the chief bits, leaving five bits in the V.2 API and two bits in the X.0 API free for future use.

[2] In the X.0 API the thread ID is 32 bits wide and only consists of a low word.

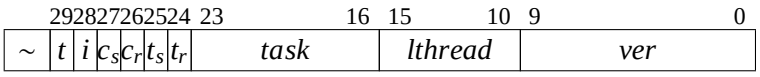


Figure 4.4: Thread ID (X.0)

Figure 4.5 summarizes the meaning of the redefined chief bits. The bits t_r and t_s specify whether the kernel should interpret the send and receive timeouts contained in the timeout descriptor of the *ipc* system call as absolute or relative timeouts. If the user specified the send or receive timeout as an absolute timeout, the bits c_r and c_s contain the *clock* bit of the absolute timeout as described in Section 3.7. If a thread sets the *i*-bit during an *ipc* system call, it signals the kernel that the system call should not return until the thread's next period has begun. The *t*-bit allows a thread to specify which sender role of the destination thread it wants to rendezvous with. A set *t*-bit selects the preemption sender role, whereas a clear *t*-bit selects the thread's regular sender role. By setting the *t*-bit a preempter can, for example, perform a closed-wait for Preemption-IPC messages from a specific thread. By evaluating the *t*-bit a preempter can also distinguish between regular IPC messages from a thread and Preemption-IPC messages sent by the kernel on behalf of the preempted thread.

Bit	Function	Bit=0	Bit=1
t_r	Format of receive timeout	Relative timeout	Absolute timeout
t_s	Format of send timeout	Relative timeout	Absolute timeout
c_r	Clock bit of receive timeout	only for absolute receive timeouts	
c_s	Clock bit of send timeout	only for absolute send timeouts	
i	IPC semantics	Regular IPC	Next-Period IPC
t	Thread ID semantics	Regular TID	Preemption ID

Figure 4.5: Redefined Chief-ID Bits

The following sections cover each of the new quality-assuring-scheduling functions in detail. I will explain how the new functions integrate with the existing L4 API and how each function is implemented in the Fiasco microkernel.

4.3 Extensions to thread_schedule

The API for the *thread_schedule* system call specifies a parameter word (*param word*) and defines that bits 16 through 19 of that word should be set to zero. To multiplex several scheduling related functions on top of *thread_schedule*, I implemented a *mode* field in the parameter word, using those 4 bits. I also added a 64 bit *time* input parameter, which user applications can use to pass raw 64 bit time values to the kernel instead of using the typical mantissa/exponent encoding. Figure 4.6 shows the API for the *thread_schedule* system call. Modified or added parameters have been marked with a star (*).

Depending on the value of the new *mode* parameter the *thread_schedule* system call provides the following functions:

- Mode 0 : Modify and return regular scheduling parameters (*set_regular*)

- Mode 1 : Add a reservation scheduling context (*rt_add*)
- Mode 2 : Remove all reservation scheduling contexts (*rt_remove*)
- Mode 3 : Set period length (*rt_period*)
- Mode 4 : Begin strictly periodic execution (*rt_begin_periodic*)
- Mode 5 : Begin periodic execution with minimal interrelease times (*rt_begin_periodic*)
- Mode 6 : End periodic execution (*rt_end_periodic*)

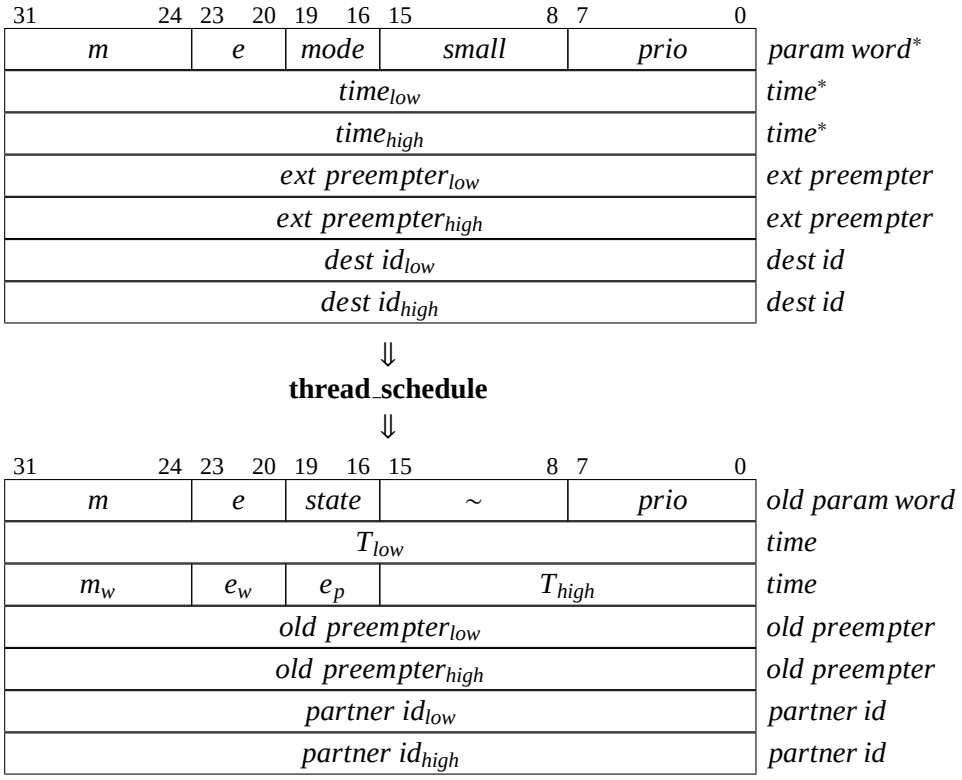


Figure 4.6: Extended API for thread_schedule

To modify a thread's scheduling parameters using *thread_schedule*, the caller must have a high enough master control priority (MCP). According to the L4 specification [19], an attempt to modify a thread's scheduling parameters succeeds if the current priority of the specified destination thread is less than or equal to the caller's MCP. With the new possibility to have multiple scheduling contexts per thread, a thread's current priority typically changes when its active scheduling context changes or when the thread receives a time donation from a different thread. The current implementation therefore interprets the specification to mean the priority of the thread's regular scheduling context, rather than the thread's current priority. I hope that with the addition of an improved rights delegation model to L4, the permission check for the modification of scheduling parameters can be refined. Note that a thread can act as its own admission server and modify its own scheduling parameters if its MCP is greater than or equal to its regular priority. For threads which are not supposed to be able to modify

their own scheduling parameters, I therefore recommend that their MCPs be set below their regular priority upon creation.

The *thread_schedule* system call fails and returns -1 in the parameter word if the destination thread does not exist or its regular priority is greater than the caller's master control priority. Several subfunctions of *thread_schedule* have additional conditions under which they fail, which I mention during the description of each function.

4.3.1 Adding Reservation Scheduling Contexts (rt_add)

To add a new reservation scheduling context to a thread, the calling thread must invoke a *thread_schedule* system call and specify the destination thread id (*dest id*), priority (*prio*) and time quantum (m, e) of the new reservation scheduling context. Additionally, the caller must set the mode field of the parameter word to 1.

The *rt_add* function fails and returns -1 in the parameter word under the following conditions:

1. the destination thread is executing in periodic scheduling mode or is transitioning to periodic mode and its deadline timeout has already been programmed to the beginning of the first period; or
2. the caller attempts to set the priority of the new scheduling context higher than its own MCP; or
3. the caller attempts to specify an infinite or zero time quantum for the new reservation scheduling context.

On success, the kernel returns 0 in the parameter word, allocates a new scheduling context for the destination thread and initializes it with the following parameters:

- **Owner**: The owner of the new reservation scheduling context is the destination thread.
- **ID**: The identification number of the new reservation scheduling context is the number of the scheduling context preceding the regular scheduling context in the destination thread's list of scheduling context plus 1. If no reservation scheduling contexts exist, the regular scheduling context precedes itself.
- **Prio**: The priority of the new reservation scheduling context is the value specified in the *prio* field of the parameter word.
- **Quantum**: The time quantum of the scheduling context is encoded in the mantissa m and exponent e fields of the parameter word. The kernel computes the absolute value of the time quantum as $t = m \cdot 4^{15-e}\ \mu s$ and rounds it up to the next higher value supported by the granularity of the kernel timer.
- **Left**: The remaining time quantum is equal to the total time quantum.
- **Preemption Time and Type**: The newly created reservation scheduling context has no pending preemption event.
- **Prev, Next**: The kernel inserts the new scheduling context directly before the regular scheduling context in the thread's list of scheduling contexts.

User applications must set up reservation scheduling contexts one by one. There is no possibility of setting up multiple reservation scheduling contexts with a single system call. The kernel assigns thread local numbers to reservation scheduling contexts in order of their creation. Once set up, the parameters of a reservation scheduling context cannot be modified.

4.3.2 Removing Reservation Scheduling Contexts (rt_remove)

For the removal of reservation scheduling contexts of a thread, the caller must invoke a *thread_schedule* system call, specify the destination thread id (*dest id*) and set the mode field of the parameter word to 2.

The current implementation removes all of a thread's reservation scheduling contexts at once, because I could not find a scenario where it would be useful to remove single reservation scheduling contexts. Typically an admission server will either leave a thread's schedule unchanged or modify it completely. Disallowing the removal of single scheduling contexts has the added benefit that the numbering of the scheduling contexts does not change at runtime.

The *rt_remove* function fails and returns -1 in the parameter word if the destination thread is executing in periodic scheduling mode or is transitioning to periodic mode and its deadline timeout has already been programmed to the beginning of the first period.

On success, the kernel returns 0 in the parameter word and deallocates all of the destination thread's reservation scheduling contexts.

4.3.3 Setting Period Length (rt_period)

Setting the period length of a thread sets the time interval after which the kernel will restart a thread's schedule by activating the first reservation scheduling context of that thread. If the thread has no reservation scheduling contexts, the kernel reactivates the thread's regular scheduling context. The period length of a thread is only meaningful for threads that are executing in periodic scheduling mode.

To set the period length of a thread, the caller must invoke a *thread_schedule* system call, specify the destination thread id (*dest id*) and set the mode field of the parameter word to 3. The caller passes the new period length (*time*) in microseconds to the kernel as an absolute 64 bit value. The kernel rounds the period length up to the next higher value supported by the granularity of the kernel timer.

This function always succeeds and returns 0 in the parameter word, because the caller can set a new period length in any scheduling mode. However, the change does not take effect until the thread starts its next period.

4.3.4 Starting Periodic Execution (rt_begin_periodic)

Once an admission server has set up a thread's scheduling contexts and period length, it can instruct the kernel to enable periodic scheduling mode for the destination thread by executing the *rt_begin_periodic* function. The caller specifies an absolute point in time at which the destination thread shall begin periodic execution. The destination thread must signal that it is ready to enter periodic mode by performing an *rt_next_period* call by the time the periodic execution shall begin. If the destination thread fails to do so, the transition to periodic mode fails.

To enable periodic scheduling mode, the caller must invoke a *thread_schedule* system call, specify the destination thread id (*dest id*) and set the mode field of the parameter word to 4 for a strictly periodic thread, or to 5 for a periodic thread with minimal interrelease times. Additionally, the caller must specify the beginning of the destination thread's first period (*time*).

The *rt_begin_periodic* function fails and returns -1 in the parameter word if the destination thread is already executing in periodic scheduling mode or is transitioning to periodic mode and its deadline timeout has already been programmed to the beginning of the first period.

On success the kernel sets the type bit in the scheduling mode field of the destination thread to 0 (strictly periodic) or 1 (periodic with minimal interrelease times), programs the thread's deadline timeout to the beginning of the thread's first period, and returns 0 in the parameter word.

4.3.5 Stopping Periodic Execution (rt_end_periodic)

If a periodic thread has finished its periodic execution, it can request that the admission server execute the *rt_end_periodic* function.

To disable periodic scheduling mode, the caller must invoke a *thread_schedule* system call, specify the destination thread (*dest id*), and set the mode field of the parameter word to 6.

The *rt_end_periodic* function fails and returns -1 if the thread is executing in conventional scheduling mode and is not transitioning to periodic mode. If the thread is executing in periodic mode or a transition to periodic mode is in progress, the function succeeds and returns 0. The kernel aborts an ongoing transition to periodic mode, resets the thread's scheduling mode to conventional mode and cancels the thread's deadline timeout. If the thread's active scheduling context is not the thread's regular scheduling context, the kernel deactivates the reservation scheduling context that is active and activates the thread's regular scheduling context. Additionally the kernel clears the thread's `Thread_delayed_deadline` bit and wakes the thread up if it had been waiting for the beginning of its next period.

4.4 Extensions to thread_switch

The API for *thread_switch* provides many free registers for extensions and only specifies a destination thread parameter (*dest id*). The L4 reference manual [19] defines that the system call performs the following actions depending on the destination id:

- **dest id = nil**
  The calling thread voluntarily releases the current scheduling context[3] and invokes the scheduler to select a new current scheduling context. The scheduler also selects a new current thread.

- **dest id ≠ nil**
  If the specified destination thread is ready, the calling thread donates the current scheduling context to the destination thread and that thread becomes the current thread. If the destination thread is not ready, then the system call acts like the former case.

[3] This is often called "yield"

The manual does not distinguish between the thread's own scheduling contexts and scheduling contexts donated to the thread by other threads. In my opinion it is perfectly reasonable for a thread to further donate a scheduling context that it does not own, thus forming a transitive donation chain. However, I consider it a design flaw that a thread may release scheduling contexts of other threads. Because a thread typically does not know whether it is running on its own or a donated scheduling context, it cannot decide whether it is yielding its own scheduling context as intended, or ruining another thread's reservation scheme by yielding a donated scheduling context, which activates the next reservation scheduling context of the owner thread.

The current implementation makes the behaviour of *thread_switch* deterministic. If the calling thread specifies *dest id = nil*, then it releases its own active scheduling context; if it specifies *dest id ≠ nil*, it donates the current scheduling context to the specified destination thread. I extended the API for *thread_switch* with several new parameters as shown in Figure 4.7. The added parameters are marked with a star (*).

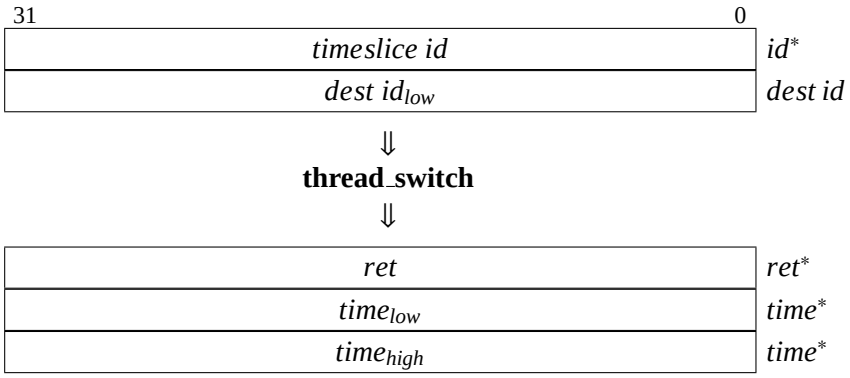


Figure 4.7: Extended API for thread_switch

4.4.1 Releasing Reservation Scheduling Contexts (rt_next_reservation)

Due to the refined behaviour of the *thread_switch* system call, implementing the functionality of releasing a thread's reservation scheduling context was straightforward. The calling thread must specify the destination thread (*dest id*) as *nil id* and then perform a *thread_switch* system call. This automatically achieves the desired effect of activating the thread's next scheduling context.

However, for reservation scheduling contexts there is a possible race condition between a voluntary release and a kernel enforced release of the active scheduling context due to a depleted time quantum. Consider the following sequence of actions:

1. A periodic thread has finished computing an optional part of the work on its third reservation scheduling context and the time quantum of that scheduling context is almost depleted.

2. The periodic thread wants to start working on the next optional part and therefore calls the *rt_next_reservation* function to release its third and activate the fourth reservation scheduling context.

3. Meanwhile the time quantum on the third scheduling context is depleted and the kernel preempts the periodic thread, sends a Preemption-IPC message to the thread's preempter, and automatically activates the thread's fourth reservation scheduling context.

4. The next time the periodic thread resumes execution on its fourth reservation scheduling context, the kernel handles the *rt_next_reservation* call and switches to the thread's fifth reservation scheduling context.

Such a situation causes the thread to compute parts of its work on wrong reservation scheduling contexts and disturbs the thread's execution. The current implementation solves this problem by letting the calling thread specify which scheduling context it thinks it is executing on. The thread passes the identification number (*timeslice id*) of its alleged active scheduling context as parameter of the *thread_switch* system call. If the thread and the kernel agree on what the thread's active scheduling context is – that is if the active scheduling context did not change meanwhile – then the kernel releases the active scheduling context and activates the next scheduling context.

Otherwise the system call fails with a return value (*ret*) of -1 and the active scheduling context does not change. On success the system call returns 0 and sets the *time* return value to the remaining time quantum of the released scheduling context. The value resembles a 64 bit time quantum in microseconds.

4.5 Extensions to ipc

For periodic threads the beginning of a new period differs, depending on the type of thread:

- If the thread is *strictly periodic*, the end of one period is equivalent to the beginning of the next period. The interrelease times of all jobs of a strictly periodic thread are constant.
- If the thread is *periodic with minimal interrelease times*, the end of one period permits the beginning of the next period. However, the next period does not start until a periodic event releases the next job. The event can be the occurrence of an interrupt, the expiration of a timer, or the reception of a message. The interrelease times of all jobs are not constant, but the thread's period is their minimum.

To be able to handle both types of periodic threads in the Fiasco microkernel, the implementation of the *rt_next_period* function must therefore allow a thread to wait for the beginning of its next period in combination with waiting for a specific event. Fortunately L4 maps almost all events to IPC messages, so I implemented *rt_next_period* on top of the *ipc* system call. Nevertheless the implementation must provide a possibility for strictly periodic threads to wait only for the beginning of the next period, without engaging in an IPC. The following section describes the implementation of *rt_next_period* for both types of periodic threads.

4.5.1 Waiting for the Next Period (rt_next_period)

When the kernel begins a new period for a periodic thread, it programs the thread's deadline timeout to trigger at the thread's deadline, in order to detect if the thread misses the deadline. The expiration of the deadline timeout also signals that the thread's current period is over and that the new period can begin if all preconditions are satisfied.

Strictly Periodic Thread

When a strictly periodic thread performs an *ipc* system call with the *i*-bit set to 1 in the destination thread id, as described in Section 4.2, it signals the kernel that this system call also waits for the beginning of the thread's next period. A strictly periodic thread typically does not wait for the next period in combination with waiting for an IPC event, although the current implementation permits it. If a strictly periodic thread wants to wait for the beginning of the next period without sending or receiving anything, it can skip the send and receive phase of the *ipc* call by specifying itself as the destination thread. I call an IPC that contains neither a send nor a receive phase a *Null-IPC*.

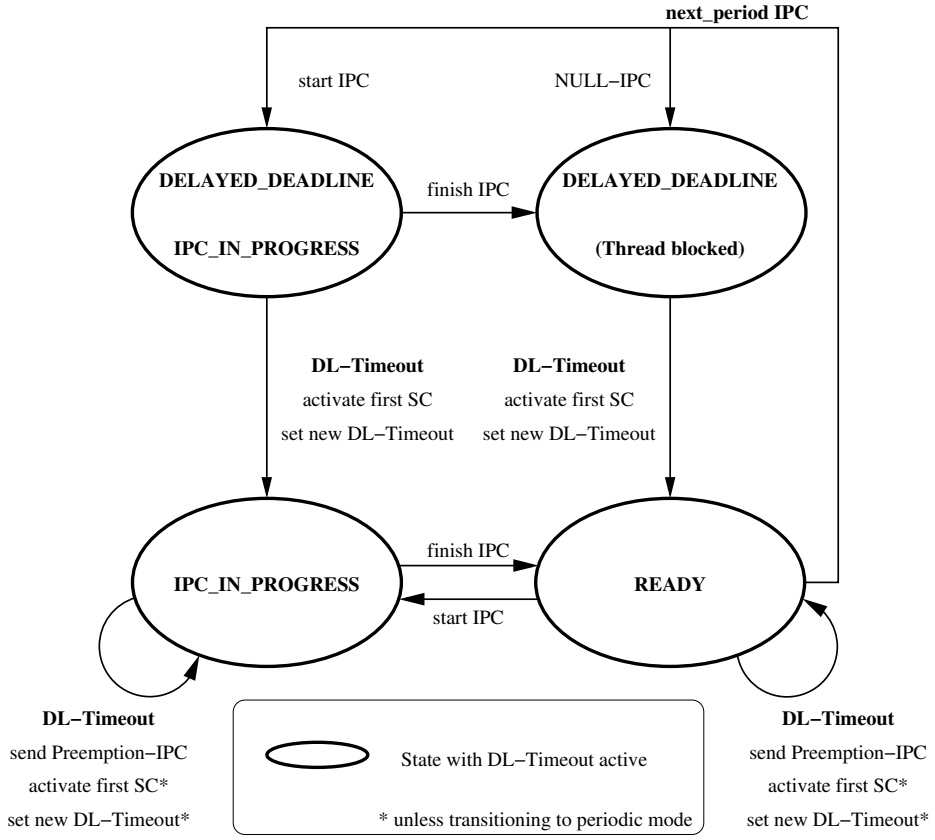


Figure 4.8: State Transition Chart for rt_next_period (strictly periodic thread)

Figure 4.8 shows a state transition chart for an *ipc* system call with next-period semantics in periodic scheduling mode. When a thread in *ready* state enters the kernel during the *ipc* system call, the kernel recognizes the set *i*-bit and sets the `Thread_delayed_deadline` bit in the thread's status word. The kernel prevents the thread from exiting the kernel as long as the `Thread_delayed_deadline` bit is set. Depending on whether the system call was a Null-IPC or an IPC with a send or receive phase, the thread either immediately blocks because it cannot leave the kernel, or it attempts to engage in an IPC with the specified partner. If the partner does not respond in a timely manner, the rendezvous may fail with a send or receive timeout. If the rendezvous succeeds, the kernel additionally sets the `Thread_ipc_in_progress` bit in thread's status word.

If the thread performed a Null-IPC, it falls through to the bottom of the *ipc* function. Otherwise the thread can perform exactly one IPC with another thread until it also arrives at the bottom of the *ipc* function. Shortly before the *ipc* call returns to user-mode, it executes the following code fragment:

```
while (state_change_safely (~(Thread_delayed_deadline | Thread_ready),
                            Thread_delayed_deadline))
  {
    schedule();
  }
```

The loop tests if the threads's `Thread_delayed_deadline` bit is set and, if it is, puts the thread to sleep by removing the `Thread_ready` bit in the status word, then invoking the scheduler. The thread gets past the loop once its deadline timeout triggers, to indicate that the current period is over. The handler for the expiration of the deadline timeout clears the `Thread_delayed_deadline` bit, restarts the periodic schedule by activating the thread's first reservation scheduling context (*activate first SC*), and reprograms the deadline timeout to the end of the following period (*set new DL Timeout*). If the periodic thread did not perform a Null-IPC and is still in its send or receive phase at the time the kernel clears the `Thread_delayed_deadline` bit, then the thread's next period begins nonetheless. User applications can use this feature to perform exactly one IPC per period.

If the function that handles the expiration of a thread's deadline timeout finds that the `Thread_delayed_deadline` bit is not set in the status word, it knows that the periodic thread did not invoke an *rt_next_period* call in a timely manner and therefore missed its deadline. In that case it sends a Preemption-IPC message to the thread's preempter. If the thread was transitioning to periodic mode, the transition fails. Otherwise the kernel starts the thread's next period.

Periodic Thread with Minimal Interrelease Times

The implementation of *rt_next_period* for periodic threads with minimal interrelease times is slightly more complicated because a new period can begin only if both of the following conditions are satisfied:

1. The deadline of the current period has passed, indicating that the current period is over.
2. The periodic event has occurred. For the kernel this means that the send and receive phases of the *ipc* call have been finished.[4]

The kernel handles the first condition similarly to strictly periodic threads. When a periodic thread enters the kernel due to a next-period IPC, the kernel sets the `Thread_delayed_deadline` bit. The expiration of the deadline timeout clears that bit. For the second condition the kernel uses a second bit in the thread's status word, `Thread_delayed_ipc`, which it sets together with the `Thread_delayed_deadline` bit on kernel entry. The kernel clears that bit when an IPC associated with the *rt_next_period* call completes.

For a periodic thread with minimal interrelease times the next period cannot begin until the `Thread_delayed_deadline` bit and the `Thread_delayed_ipc` bit have both been cleared.

Figure 4.9 shows the state transition diagram for periodic threads with minimal interrelease times. Because the beginning of the next period is always bound to the occurrence of a periodic event, the

[4] Either the send phase or the receive phase can be omitted, but not both.

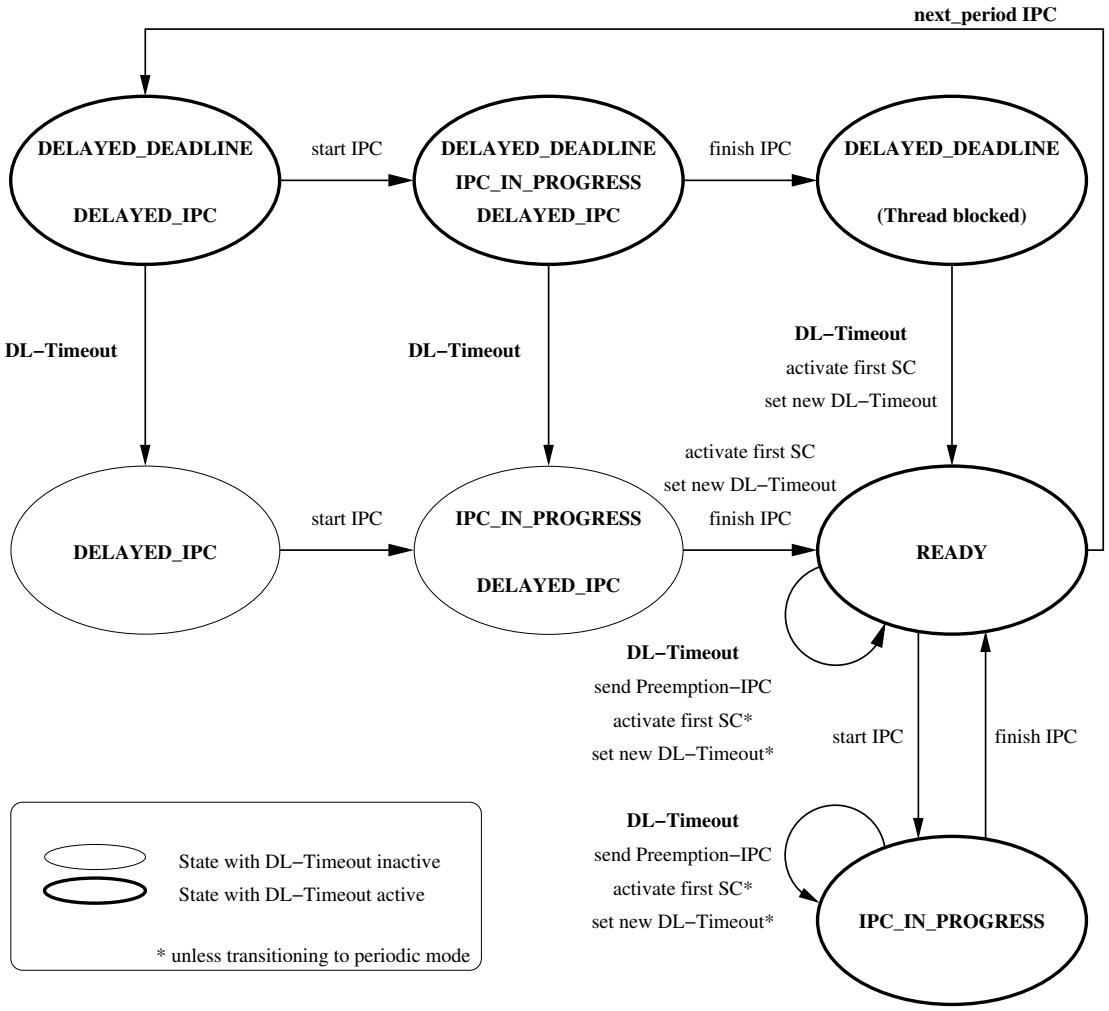


Figure 4.9: State Transition Chart for rt_next_period (periodic thread)

kernel does not permit the *rt_next_period* function as Null-IPC for periodic threads with minimal interrelease times.

The *rt_next_period* call for periodic threads with minimal interrelease times behaves similarly to that for strictly periodic threads; the handling of mode transition errors and deadline misses is exactly the same. The fundamental difference, apart from the additional `Thread_delayed_ipc` bit, is the point in time when the kernel starts the thread's next period.

- If the send and receive phases of the IPC resembling the periodic event finish before the old period is over, then the kernel clears the `Thread_delayed_ipc` bit before the `Thread_delayed_deadline` bit. In that case the function that handles the expiration of the thread's deadline timeout recognizes that the `Thread_delayed_ipc` has already been cleared. It then restarts the periodic schedule by activating the thread's first reservation scheduling context (*activate first SC*), and reprograms the deadline timeout to the end of the next period (*set new DL Timeout*).

- If the deadline timeout triggers before the send and receive phases of the IPC finish, then the kernel clears the `Thread_delayed_deadline` bit before the `Thread_delayed_ipc` bit. In that case the function that finishes the IPC recognizes that the `Thread_delayed_deadline` bit has already been cleared. It then restarts the periodic schedule by activating the thread's first reservation scheduling context (*activate first SC*), and reprograms the deadline timeout to the end of the next period (*set new DL Timeout*).

Note that the latter event starts the next period after it detects that the former event has already occurred. When the former event finds that the latter event has not yet occurred, it performs no other action than clearing the bit in the status word corresponding to the former event.

A noteworthy aspect is the fact that the completion of a next-period IPC can actually start the next period for two threads with minimal interrelease-times: namely, if the periodic event for the first thread was the reception of an IPC message from the second thread, and the periodic event for the second thread was the ability to send that message to the first thread. This scenario can only occur if the second thread performs a send-only next-period IPC.

The current implementation of *rt_next_period* allows user applications to combine waiting for the beginning of the next period with any form of IPC that has at least a send phase or a receive phase. The design provides maximum flexibility and includes the ability to abort the next-period IPC using absolute or relative timeouts for the send and receive phases. An IPC returning an error code, such as a send timeout, to the caller loses its next-period semantics and does not delay the thread until the beginning of the next period.

Chapter 5

Performance

5.1 Performance of the Ready List

For the implementation of interrupt handlers with little overhead it was important to optimize the structure of the ready list to facilitate fast enqueue operations. When a timer interrupt triggers the expiration of an IPC timeout, the kernel wakes up the corresponding thread and enqueues it in the ready list. Similarly, if a device interrupt occurs and wakes up an IRQ thread attached to that interrupt, the kernel must also enqueue that IRQ thread in the ready list. The redesigned ready list has the following performance properties:

- **Enqueueing of threads**:
  The kernel can always enqueue a thread in the ready list with an O(1) time complexity. If the kernel enqueues a thread in a priority ring which already contains other threads with the same priority, the kernel simply moves the thread to the end of the ring by enqueueing it before `prio_next` of that priority ring. If the thread is the first thread to be inserted into an empty priority level, the kernel only needs to point `prio_next` of that priority level to the thread, which then forms a ring consisting only of itself.

- **Dequeueing of the last thread of the highest priority**:
  When the kernel dequeues the last thread of the highest priority, it has to find the next lower populated priority ring to update `prio_highest` accordingly. The kernel must iterate over the `prio_next` array starting at the old value of `prio_highest` and moving down the array until it finds a nonempty priority ring, indicated by a non-null `prio_next` field. This special case therefore has a time complexity of O(N), where N denotes the number of priority levels supported by the system.

- **Dequeueing of all other threads**:
  All other dequeue operations have a time complexity of O(1). When the kernel removes a thread from its priority level, it links the thread's predecessor and successor together to close the gap in the priority ring. If the thread was the last thread of that priority in the ready list, the kernel must additionally set `prio_next` for that priority to null. An update of `prio_highest` is not necessary.

5.2 Microbenchmarks

This section examines the performance of the Fiasco scheduler and the extensions for quality-assuring scheduling using microbenchmarks. For each benchmark I specify the elapsed time in wall clock time and in processor cycles, as measured by the CPU's `rdtsc` instruction [11] – excluding the overhead of said instruction.

I conducted the benchmarks using three different x86-based hardware platforms. Figure 5.1 gives an overview of the test machines with their different processor types and clock speeds. The last column shows the overhead for a transition from user mode to kernel mode and back. I measured the system call cost using an interrupt gate and a handler that saves user state, restores it and then returns to user mode.

CPU	CPUID	Clock Speed	Kernel Entry and Exit	
Intel Pentium 4	F:0:A	1600 MHz	1528 cycles	955 ns
Intel Celeron	6:8:A	900 MHz	288 cycles	320 ns
AMD Duron	6:3:1	800 MHz	210 cycles	263 ns

Figure 5.1: Benchmarked Systems

The first benchmark (Figure 5.2) measures the time the kernel needs to invoke the scheduler, without switching to a different scheduling context or a different thread. In the test scenario the highest priority thread repeatedly calls `schedule()`, even though the ready list does not change. Therefore the scheduler always reselects the same current scheduling context and the same current thread. The majority of the overhead results from acquiring and releasing the CPU lock for the duration of the dispatch operation and from checking the next thread to run in the highest populated priority ring of the ready list, to find that nothing needs changing. The benchmark illustrates the minimal additional cost users would have to pay for each message transfer if the function to release a wait-free lock were to generally invoke the scheduler to determine the next thread to run instead of maintaining the lock_helper field in the TCB and switching directly to the donating thread. An L4 IPC call consists of two message transfers; during each message transfer the sender locks down the receiver and releases the lock afterwards.

Pentium 4 (1600 MHz)		Celeron (900 MHz)		Duron (800 MHz)	
CPU Cycles	Wall Clock	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock
200	125 ns	150	167 ns	71	88 ns

Figure 5.2: Scheduler Invocation Overhead

The second benchmark (Figure 5.3) measures the time for a context switch in the kernel, including the time required to save the CPU state of the old thread and to load the CPU state of the new thread. The numbers were obtained using warm caches and do not include the overhead of saving and restoring the FPU state, because the Fiasco microkernel implements a lazy FPU switching scheme that does not save the FPU state on each context switch. The first line shows the cost resulting from a context switch to a thread in the same address space. The second line presents the numbers for switching

to a thread in a different address space; the additional costs result from loading the new page table hierarchy, flushing the translation lookaside buffer (TLB) and from the resulting TLB refills.

Address Space	Pentium 4 (1600 MHz)		Celeron (900 MHz)		Duron (800 MHz)	
	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock
Same	136	85 ns	117	130 ns	78	98 ns
Different	440	275 ns	211	235 ns	183	229 ns

Figure 5.3: Context Switch Overhead Depending on Destination Address Space

Figure 5.4 illustrates the overhead of the timer interrupt handler depending on the number of timeouts that have expired at the time the interrupt occurs. For this benchmark I used the APIC as timer source and created up to 128 threads in the same address space. Each thread sleeps using the *ipc* system call in combination with an absolute timeout to ensure that it wakes up at exactly the same point in time as the other threads. The numbers only reflect the duration of the in-kernel handler and do not include the overhead of the kernel entry and exit resulting from the interrupt, which is listed in Figure 5.1.

Timeouts expired	Pentium 4 (1600 MHz)		Celeron (900 MHz)		Duron (800 MHz)	
	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock
0	272	170 ns	135	150 ns	40	50 ns
1	696	435 ns	656	729 ns	308	385 ns
2	744	465 ns	978	1.1 μs	506	633 ns
4	1548	968 ns	1279	1.4 μs	934	1.2 μs
8	2260	1.4 μs	2053	2.3 μs	1476	1.8 μs
16	3308	2.1 μs	5406	6.0 μs	4863	6.1 μs
32	8804	5.5 μs	9186	10.2 μs	11454	14.2 μs
64	12720	8.0 μs	48687	54.1 μs	49993	62.5 μs
128	56256	35.2 μs	84903	94.3 μs	84775	106.0 μs

Figure 5.4: Timer Interrupt Handler Overhead Depending on the Number of Expired Timeouts

The next benchmark measures the overhead of the in-kernel timer interrupt handler depending on which hardware timer is used. The overhead includes acknowledging the interrupt at the PIC and advancing the kernel clock. During this benchmark the timer interrupt did not expire any timeouts and did not invoke the scheduler. I measured the overhead for the APIC in periodic mode (P) and in one-shot mode (O). The extra cost for one-shot APIC timer interrupts results from the necessity to synchronize the kernel clock with the TSC[1]. For periodic timers it suffices to add the period of the timer interrupt to the kernel clock.

5.3 System-Call Performance

Figure 5.6 shows the performance of the new quality-assuring scheduling functions implemented during this work. The measurements were done from user mode and include the costs for the system-

[1] Time Stamp Counter

Timer & Mode	Pentium 4 (1600 MHz)		Celeron (900 MHz)		Duron (800 MHz)	
	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock
APIC P	272	170 ns	135	150 ns	40	50 ns
APIC O	816	510 ns	425	472 ns	200	250 ns
PIT P	1484	928 ns	744	826 ns	1343	1.7 μs
RTC P	22096	14 μs	9365	10.4 μs	18861	23.6 μs

Figure 5.5: Timer Interrupt Handler Overhead Depending on the Hardware Timer

call binding and the kernel entry and exit. For each function I list the average number of cycles with warm caches, which can vary slightly due to hardware effects. The figure lists two different rows for the *rt_add* system call. The first row, labelled "reuse", shows the overhead for allocating a new scheduling context from an existing slab page. The second row, labelled "alloc", includes the additional overhead of allocating a new memory page for a new scheduling context, which happens when the previous slab page is full. Because a scheduling context is 48 bytes in size, the kernel can fit 85 scheduling contexts on one memory page of the slab allocator.

Pentium 4 (1600 MHz)		Celeron (900 MHz)		Duron (800 MHz)	
CPU Cycles	Wall Clock	CPU Cycles	Wall Clock	CPU Cycles	Wall Clock
rt_add		**rt_add**		**rt_add**	
reuse 2588	1.6 μs	1192	1.3 μs	768	960 ns
alloc 16000	10.0 μs	10000	11.1 μs	10000	12.5 μs
rt_period		**rt_period**		**rt_period**	
2056	1.3 μs	717	797 ns	597	746 ns
rt_begin_periodic		**rt_begin_periodic**		**rt_begin_periodic**	
2196	1.4 μs	790	878 ns	696	870 ns
rt_end_periodic		**rt_end_periodic**		**rt_end_periodic**	
2272	1.4 μs	1036	1.2 μs	640	800 ns
rt_next_reservation		**rt_next_reservation**		**rt_next_reservation**	
2580	1.6 μs	1027	1.1 μs	654	818 ns

Figure 5.6: Performance of the Quality-Assuring-Scheduling Functions

Chapter 6

Conclusion and Summary

The scheduler that I developed during this thesis has shown that the quality-assuring-scheduling model can be implemented in the Fiasco microkernel without a noticeable performance overhead, as long as the kernel does not employ dependency tracking for time donation.

The performance overhead of having multiple scheduling contexts per thread is one extra memory reference, to determine the thread's active scheduling context. The pointer to the active scheduling context is located in the same cache line as the thread's regular scheduling context, so that no additional cache or TLB misses occur for non-real-time threads.

The discussion in Section 3.9 has shown that time donation cannot be implemented without seriously impairing the performance of the system. I tend to believe that there is no implementation of time donation dependency tracking that does not either slow down IPC or introduce additional interrupt latency.

Future Work

Efficient Time Donation

Although we know how to combine scheduling and IPC to avoid priority inversion, we have not yet found a viable solution for an implementation of time donation that not degrade the performance of the kernel. Future work in this area should focus on solving the following two key problems:

1. **Handling of dead chains in the ready list**:
   In Section 3.9.1 I explained how the removal of dead donation chains from the ready list requires parts of these chains to be atomically reenqueued when a donation link inside such a donation chain breaks. Therefore it seems necessary to keep dead donation chains permanently enqueued in the ready list. To be able to dispatch threads quickly during scheduler invocations, we have to find a mechanism that allows us to mark dead donation chains and skip them in consecutive scheduler runs. We should also explore the possibility of reducing the chain traversal overhead by introducing shortcut links that lead directly from a scheduling context to the thread to which that scheduling context has most recently been donated.

2. **Efficient recalculation of the boost priority**:
   For the implementation of the stack-based priority ceiling protocol and correct accounting

mechanisms in upward-donation scenarios, the kernel has to temporarily boost the priority of scheduling contexts that are donated from a lower-priority thread to a higher-priority thread. In Section 3.9.2 I illustrated how the recalculation of the boost priority requires traversal of the donation chain starting from the owner of the scheduling context to the thread which runs on that scheduling context. For performance reasons we cannot afford to do this recalculation in the IPC path each time a donating IPC finishes, because the resulting cache and TLB misses would seriously harm the performance of microkernel-based systems.

I identified a performance problem in the handling of expired timeouts in Section 5.2. Because one instance of the timer interrupt can theoretically dequeue hundreds of timeouts at once, the timer interrupt is subject to unbounded interrupt latency. I have provoked and investigated this effect and shown the resulting latency degradation in Figure 5.4. The handler for the timer interrupt is not preemptible, because it modifies the timeout list and enqueues woken-up threads in the ready list. The handler should at least be preemptible by other hardware interrupts. We should also explore whether we can allow nesting of timer interrupts if the inner nesting levels do not process the timeout list.

Timer-Interrupt Mitigation

In Section 3.5 I explained how the APIC hardware timer in combination with one-shot timer interrupts can be used to provide fine-grained timeouts in the microsecond range. However, each one-shot timer interrupt causes additional overhead as shown in Figure 5.5. In the future we should explore if we can mitigate the interrupt overhead by handling multiple timeouts with a single one-shot interrupt. This can be achieved by coupling the wakeup time of each timeout with a granularity, such that the kernel is allowed to trigger the timeout in a certain interval instead of having to trigger it at the precise wakeup time. However, such a feature requires that users of such timeouts specify the granularity of the timeout in addition to the wakeup time.

Encoding of Absolute Timeouts

The encoding of absolute timeouts in the current implementation is not optimal, because the clock bit and the timeout-format bit reside in the chief number of the destination thread ID instead of being part of the timeout descriptor. This issue has already been addressed in the L4 specification for the X.2 API [15]. The new timeout descriptor format is shown in Figure 6.1.

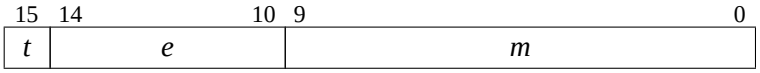


Figure 6.1: Timeout Format (X.2)

The *t*-bit denotes the type of timeout. If set to 1, the timeout is an absolute timeout; otherwise, the timeout is relative. The remaining 15 bits form the exponent *e* and the mantissa *m*. The computation of the timeout value is similar to that explained in Section 3.7. For absolute timeouts, the X.2 API encodes the clock bit as the least significant bit of the exponent, thereby reducing the size of the remaining exponent field to 4 bits.

Quality-Assuring Scheduling in Future L4 APIs

The extensions for quality-assuring scheduling should be specified in future versions of the L4 API. This includes the support for periodic threads with the accompanying rt_add, rt_remove, rt_period, rt_begin_periodic, rt_end_periodic, rt_next_reservation and rt_next_period functions as described in Chapter 4. A role bit in the thread ID could provide the distinction between IPC sender role and Preemption sender role instead of locating the bit in the chief number. The bit to specify that an IPC waits for the beginning of the next period could be relocated to the timeout descriptor.

Summary

The implementation of the quality-assuring-scheduling model in the Fiasco microkernel is a great step forward towards using the kernel for the construction of flexible real-time systems. Although the initial goal of my thesis was the implementation of a working prototype of the model, the resulting scheduler can be classified as a fully-fledged reference implementation. The extensions to the scheduling interface facilitate the use of user-level admission servers that can enforce a user-defined schedule with the help of the microkernel. The Preemption-IPC feedback mechanism that was developed and implemented allows these admission servers to react to exceptional scheduling conditions. I optimized the structure and performance of the ready list, which is a crucial condition precedent to the implementation of a scheduler that can make dispatch decisions quickly. Furthermore, I have introduced absolute timeouts as a time basis for periodic threads. In combination with one-shot timer interrupts introduced as part of this work, the kernel can provide fine-grained timeouts in the microsecond range to user applications. The most recent version of L^4Linux [16] already uses these fine-grained absolute timeouts.
This thesis also presents a solution to make wait-free locks work correctly in the presence of time donation without impairing the performance of these locks.

The new scheduler is already part of recent Fiasco microkernel versions and has received widespread testing. It can be considered stable and feature complete.

Bibliography

[1] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA, June 1986.

[2] Jen-Yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1173, September 1990.

[3] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison Wesley, Reading, MA, 1967.

[4] Dresden Realtime Operating System Website. URL: `http://os.inf.tu-dresden.de/drops/`.

[5] Fiasco Microkernel Website. URL: `http://os.inf.tu-dresden.de/fiasco/`.

[6] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *USENIX Winter Conference*, pages 97–114, CA, January 1994.

[7] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[8] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz`.

[9] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[10] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, TU Dresden, Fakultät Informatik, September 2002.

[11] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

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

[13] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium (RTSS)*, pages 112–122, 1985.

[14] L4 Environment Website. URL: `http://os.inf.tu-dresden.de/l4env/`.

[15] L4Ka Team. *L4 eXperimental Kernel Reference Manual, Version X.2*, 2003.

[16] Adam Lackorzynski. L^{4}Linux Porting Optimizations. Master's thesis, TU Dresden, March 2004.

[17] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[18] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

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

[20] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[21] Kwei-Jay Lin and Swaminathan Natarajan. Expressing and maintaining timing constraints in Flex. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS)*, pages 96–105, 1988.

[22] Linux Kernel Website. URL: `http://www.kernel.org/`.

[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, 1973.

[24] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[25] Alexander Warg. Software Structure and Portability of the Fiasco Microkernel. Master's thesis, TU Dresden, July 2003.

[26] Microsoft Windows Website. URL: `http://www.microsoft.com/windows/`.