HelenOS 0.2.0

design documentation

June 18, 2006

HelenOS 0.2.0

Contents

List of Figures

Chapter 1

Introduction

HelenOS is a portable operating system with support for a variety of modern processor architectures[1].

This book describes the design and principles of the HelenOS operating system from the perspective of its microkernel as well as from the perspective of its userspace drivers and server tasks. Its primary goal is to present ideas behind each subsystem and highlight things that are specific to HelenOS. Although this text contains references to source code (e.g. function names), these are provided only to improve reader's orientation when reading the code. This book does not attempt to be a substitute for a reference manual and the reader is strongly encouraged to look for interface details there.

1.1 How to Read This Book

Chapter 2 contains overview of the overall HelenOS architecture.

Chapter 3 describes essential data structures used both in the kernel and in the userspace.

Chapter 4 focuses on time management in the kernel and scheds some light on the userspace source of time.

Chapter 5 is dedicated to threads and the scheduling subsystem.

Chapter 7 describes memory management of physical and virtual memory.

Chapter 8 deals with the IPC subsystem.

Chapter 9 describes facilities that a userspace task can use in order to become a device driver.

Appendix A presents some architecture specific issues.

[1] amd64, ia32, ia64, mips32 and ppc32.

Chapter 2

Architecture Overview

The HelenOS operating system is designed as a relatively small microkernel assisted with a set of userspace drivers and server tasks. HelenOS is not very radical in which subsystems should or should not be implemented in the kernel - in some cases, both kernel and userspace drivers exist. The reason for creating the system as a microkernel is prosaic. Even though it is initially more difficult to get the same level of functionality from a microkernel than it is in the case of a simple monolithic kernel, a microkernel is much easier to maintain once the pieces have been put to work together. Therefore, the kernel of HelenOS, as well as the essential userspace libraries thereof can be maintained by only a few developers who understand them completely. In addition, a microkernel based operating system reaches completion sooner than monolithic kernels as the system can be used even without some traditional subsystems (e.g. block devices, filesystems and networking).

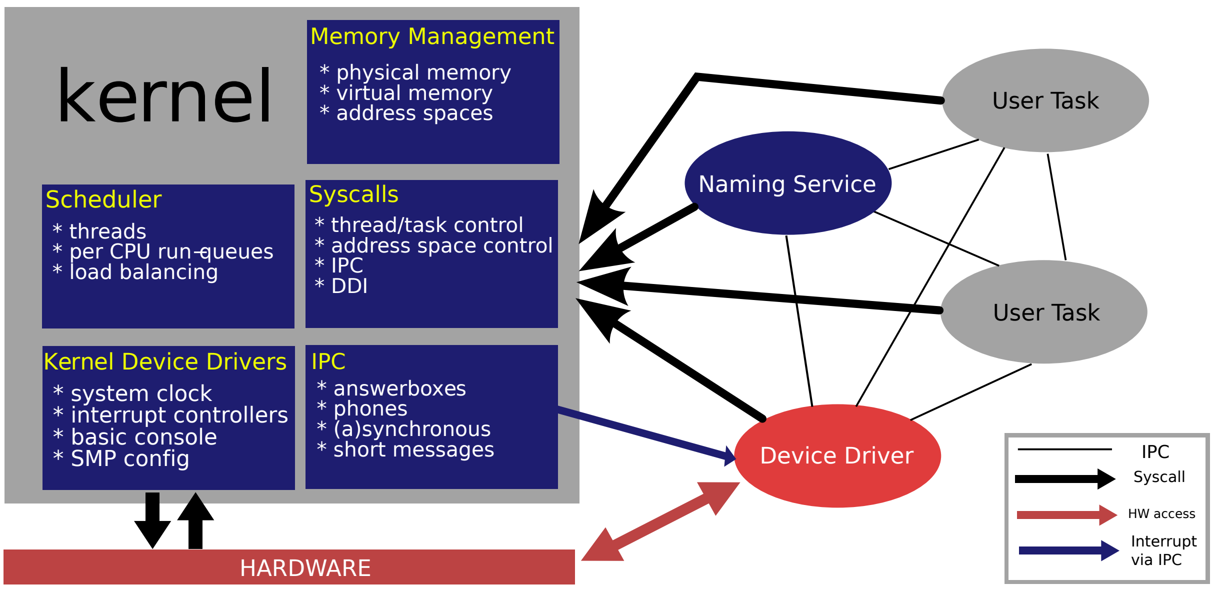

Figure 2.1: HelenOS architecture overview.

HelenOS is comprised of the kernel and the userspace server tasks. The kernel provides scheduling, memory management and IPC. It also contains essential device drivers that control the system clock and other devices necessary to guarantee a safe environment. Userspace communicates with the kernel through a small set of syscalls. The userspace layer consists of tasks with different roles, capabilities and privileges. Some of the tasks serve as device drivers, naming servers, managers of various kinds and some are just ordinary user programs. All of them communicate with other threads via kernel-provided IPC.

2.1 Scheduling

Kernel's unit of execution flow is a thread. A thread is an entity that executes code and has a stack that takes up some space in memory. The relation between kernel and userspace threads is 1:1:n, meaning that there can be several pseudo threads running within one userspace thread that maps to one kernel thread. Threads are grouped into tasks by functionality they provide (i.e. several threads implement functionality of one task). Tasks serve as containers of threads, they provide linkage to address space and are communication endpoints for IPC. Finally, tasks can be holders of capabilities that entitle them to do certain sensitive operations (e.g access raw hardware and physical memory).

The scheduler deploys several run queues on each processor. A thread ready for execution is put into one of the run queues, depending on its priority and its current processor, from where it is eventually picked up by the scheduler. Special purpose kernel threads strive to keep processors balanced by thread migration. Threads are scheduled by the round robing scheduling policy with respect to multiple priority run queues.

2.2 Memory Management

Memory management is another large subsystem in HelenOS. It serves the kernel to satisfy its own memory allocation requests, provides translation between virtual and physical memory addresses and manages virtual address spaces of userspace tasks.

Kernel allocates memory from the slab allocator, which itself allocates memory from a buddy system based allocator of physical memory frames.

The virtual address translation layer currently supports two mechanisms for mapping virtual memory pages to physical memory frames (i.e. 4-level hierarchical page tables and global page hash table), and is further extensible to other mechanisms. Userspace tasks depend on support of address spaces provided by the kernel. Each address space is a set of mutually disjunctive address space areas. An address space area is usually connected to, and backed by, anonymous memory, executable image of some program or continuous region of physical memory. However, swapping pages in and out to external memory is not supported. Address space areas can be easily shared among address spaces.

2.3 IPC

Due to the fact that HelenOS is a microkernel, strong emphasis is put on its IPC (Inter-Process Communication[1]). Tasks communicate by passing very short messages to one another or by sending (i.e. sharing) address space areas when larger data is to be transfered. The abstraction uses terms like phones, calls and answerboxes, but is similar to well-known abstraction of message queues. A task can have multiple simultaneous simplex connections to several other tasks. A connection leads from one of the source task's phones to the destination task's answerbox. The phones are used as handles for making calls to other tasks. Calls are asynchronous and can be forwarded from one task to another.

[1] The term Inter-Process Communication is slightly confusing because in HelenOS terminology there are tasks instead of processes. However, its abbreviation, IPC, is being publicly used as a standard name for similar facilities. This book will therefore use the term IPC to refer to communication among tasks.

Chapter 3

Data Structures

There is a lot of data that either flows through various HelenOS subsystems or is stored directly by them. Each subsystem uses its own data structures to represent the data. These data structures need to be kept somewhere. In order to work efficiently, HelenOS, and especially its kernel, deploys several house keeping data types that are designed to facilitate managing other data structures. Most of them serve like generic containers.

3.1 Lists

HelenOS uses doubly-circularly-linked lists to bind related data together. Lists are composed of an independent sentinel node called head and links that are always part of the object that is to be put into the list. Adding items to a list thus doesn't require any further memory allocations. Head and each link then contains forward and backward pointer. An empty list is composed of a sole head whose both pointers reference the head itself. The expense of two times bigger memory consumption as compared to memory consumption of singly linked lists is justified by constant insertion and removal times at random positions within the list.

Lists are frequently used to implement FIFO behaviour (e.g. scheduler run queues or synchronization wait queues). Contrary to the FIFO type, which is also supported by HelenOS, they don't take up any unused space and are more general. On the other hand, they are slower than in-array FIFOs and can be hardly used to implement buffers.

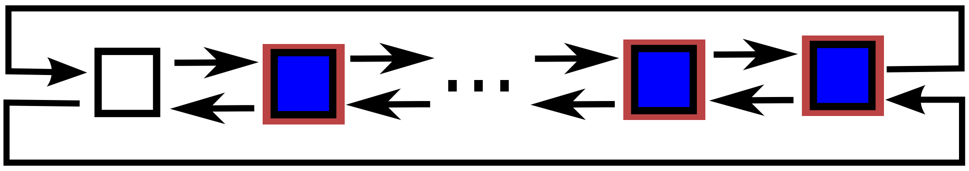

Figure 3.1: Doubly-circularly-linked list

3.2 FIFO Queues

FIFO queues are implemented as either statically or dynamically allocated arrays[1] of some generic type with two indices. The first index points to the head of the FIFO queue and the other points to the tail thereof. There can be as many items in the FIFO as is the number of elements in the array and no more. The indices are taken modulo size of the queue because as a consequence of insertions and deletions, the tail can have numericaly lower index than the head.

FIFO queues are used, for example, in ASID management code to store inactive ASIDs or in userspace keyboard driver to buffer read characters.

[1] Depending on the array size.

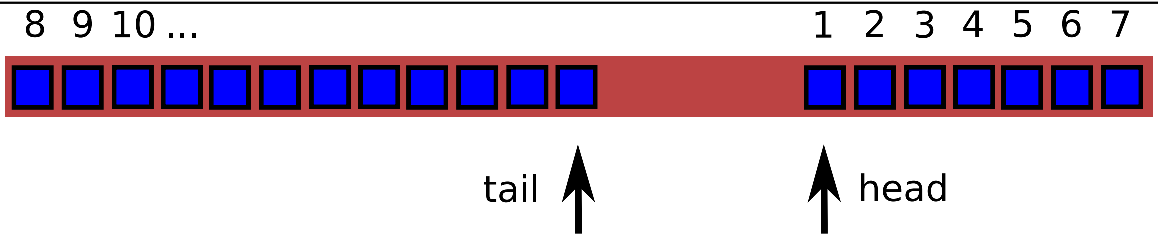

Figure 3.2: FIFO queue showing the wrap around the end of the array.

3.3 Hash Tables

The kernel, as well as userspace, provides hash table data type which uses separate chaining. The hash table type is very generic in that it forces the user to supply methods for computing the hash index, comparing items against a set of keys and the item removal callback function. Besides these virtual operations, the hash table is composed of a dynamically allocated array of list heads that represent each chain, number of chains and the maximal number of keys.

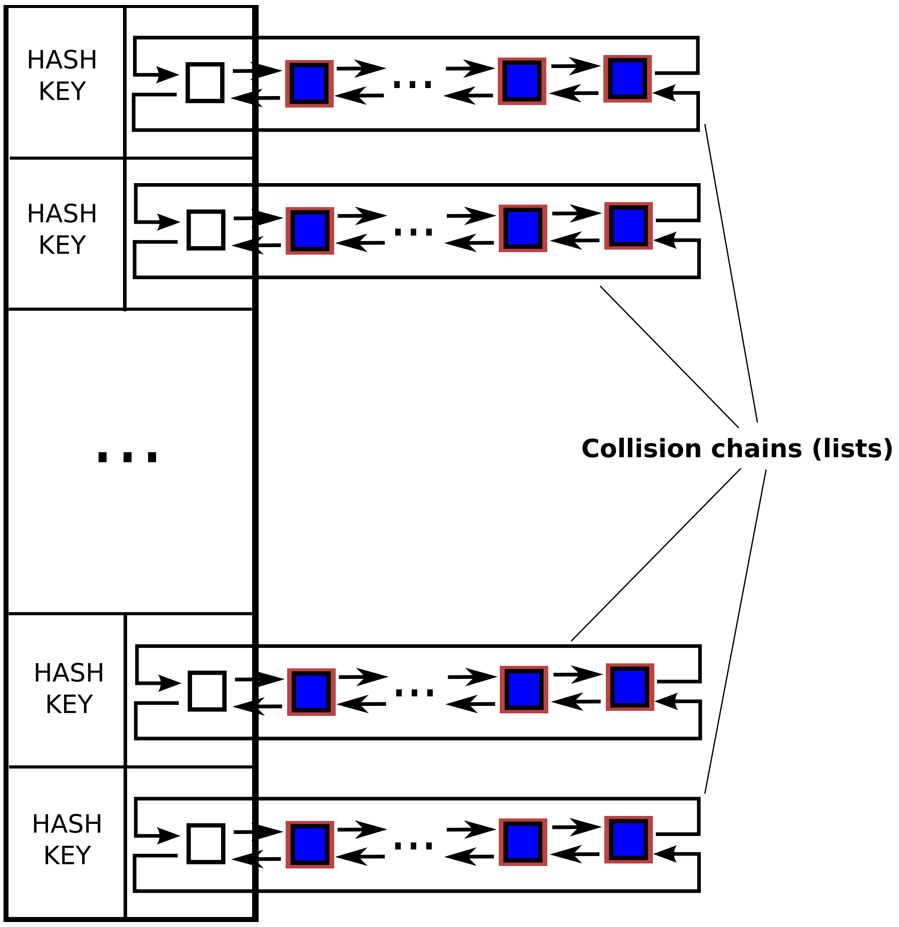

Figure 3.3: Generic hash table.

3.4 Bitmaps

Several bitmap operations such as clearing or setting consecutive bit sequences as well as copying portions of one bitmap into another one are supported.

3.5 B+trees

HelenOS makes use of a variant of B-tree called B+tree. B+trees, in HelenOS implementation, are 3-4-5 balanced trees. They are characteristic by the fact that values are kept only in the leaf-level nodes and that these nodes are linked together in a list. This data structure has logaritmic search, insertion and deletion times and, thanks to the leaf-level list, provides fantastic means of walking the nodes containing data. Moreover, B+trees can be used for easy storing, resizing and merging of disjunctive intervals.

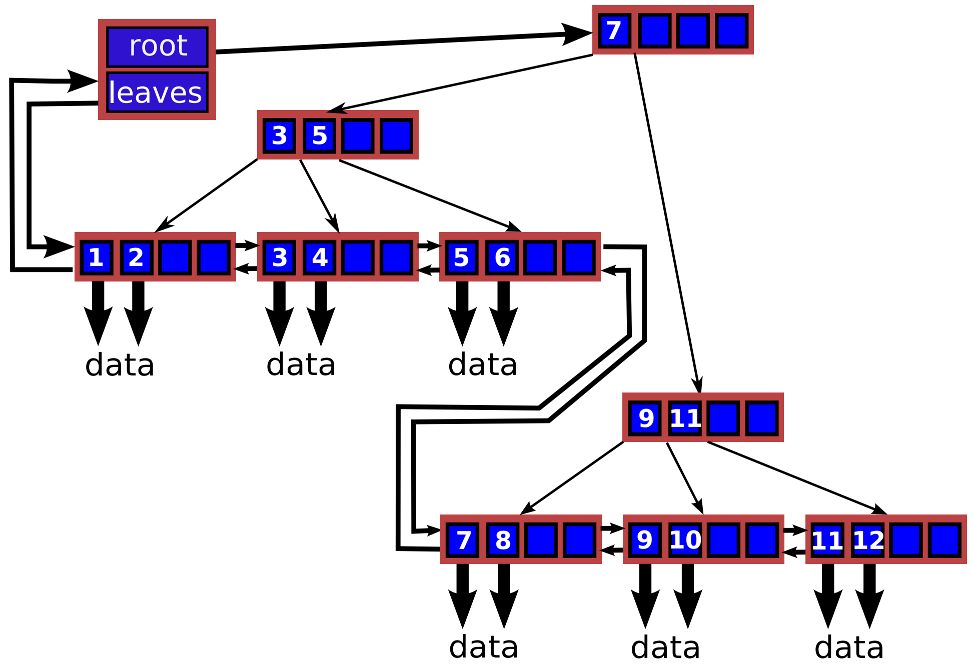

Figure 3.4: B+tree containing keys ranging from 1 to 12.

Chapter 4

Time Management

Time is one of the dimensions in which kernel as well as the whole system operates. It is of special importance to many kernel subsytems. Knowledge of time makes it possible for the scheduler to preemptively plan threads for execution. Different parts of the kernel can request execution of their callback function with a specified delay. A good example of such kernel code is the synchronization subsystem which uses this functionality to implement timeouting versions of synchronization primitives.

4.1 System Clock

Every hardware architecture supported by HelenOS must support some kind of a device that can be programmed to yield periodic time signals (i.e. clock interrupts). Some architectures have external clock that is merely programmed by the kernel to interrupt the processor multiple times in a second. This is the case of ia32 and amd64 architectures[1], which use i8254 or a compatible chip to achieve the goal.

Other architectures' processors typically contain two registers. The first register is usually called a compare or a match register and can be set to an arbitrary value by the operating system. The contents of the compare register then stays unaltered until it is written by the kernel again. The second register, often called a counter register, can be also written by the kernel, but the processor automatically increments it after every executed instruction or in some fixed relation to processor speed. The point is that a clock interrupt is generated whenever the values of the counter and the compare registers match. Sometimes, the scheme of two registers is modified so that only one register is needed. Such a register, called a decrementer, then counts towards zero and an interrupt is generated when zero is reached.

In any case, the initial value of the decrementer or the initial difference between the counter and the compare registers, respectively, must be set accordingly to a known relation between the real time and the speed of the decrementer or the counter register, respectively.

The rest of this section will, for the sake of clarity, focus on the two-register scheme. The decrementer scheme is very similar.

The kernel must reinitialize one of the two registers after each clock interrupt in order to schedule next interrupt. However this step is tricky and must be done with caution. Imagine that the clock interrupt is masked either because the kernel is servicing another interrupt or because the processor locally disabled interrupts for a while. If the clock interrupt occurs during this period, it will be pending until the interrupts are enabled again. Theoretically, it could happen an arbitrary counter register ticks later. Which is worse, the ideal time period between two non-delayed clock interrupts can also elapse arbitrary number of times before the delayed interrupt gets serviced. The architecture-specific part of the clock interrupt driver must avoid time drifts caused by such behaviour by taking proactive countermeasures.

Let us assume that the kernel wants each clock interrupt be generated every `TICKCONST` ticks. This value represents the ideal number of ticks between two non-delayed clock interrupts and has some known relation to real time. On each clock interrupt, the kernel computes and writes down the expected value of the counter register as it hopes to read it on the next clock interrupt. When that interrupt comes, the kernel reads the counter register again and compares it with the written down value. If the difference is smaller than or equal to `TICKCONST`, then the time drift is none or small and the next

[1] When running in uniprocessor mode.

interrupt is scheduled earlier with a penalty of so many ticks as is the value of the difference. However, if the difference is bigger, then at least one clock signal was missed. In that case, the missed clock signal is remembered in the special counter. If there are more missed signals, each of them is recorded there. The next interrupt is scheduled with respect to the difference similarily to the former case. This time, the penalty is taken modulo `TICKCONST`. The effect of missed clock signals is remedied in the generic clock interrupt handler.

4.2 Timeouts

Kernel subsystems can register a callback function to be executed with a specified delay. Such a registration is represented by a kernel structure called `timeout`. Timeouts are registered via *`timeout_register`* function. This function takes a pointer to a timeout structure, a callback function, a parameter of the callback function and a delay in microseconds as parameters. After the structure is initialized with all these values, it is sorted into the processor's list of active timeouts, according to the number of clock interrupts remaining to their expiration and relatively to already listed timeouts.

Timeouts can be unregistered via *`timeout_unregister`*. This function can, as opposed to *`timeout_register`*, fail when it is too late to remove the timeout from the list of active timeouts.

Timeouts are nearing their expiration in the list of active timeouts which exists on every processor in the system. The expiration counters are decremented on each clock interrupt by the generic clock interrupt handler. Due to the relative ordering of timeouts in the list, it is sufficient to decrement expiration counter only of the first timeout in the list. Timeouts with expiration counter equal to zero are removed from the list and their callback function is called with respective parameter.

4.3 Generic Clock Interrupt Handler

On each clock interrupt, the architecture specific part of the clock interrupt handler makes a call to the generic clock interrupt handler implemented by the *`clock`* function. The generic handler takes care of several mission critical goals:

- expiration of timeouts,
- updating time of the day counters for userspace and
- preemption of threads.

The *`clock`* function checks for expired timeouts and decrements unexpired timeout expiration counters exactly one more times than is the number of missed clock signals (i.e. at least once and possibly more times, depending on the missed clock signals counter). The time of the day counters are also updated one more times than is the number of missed clock signals. And finally, the remaining timeslice of the running thread is decremented with respect to this counter as well. By considering its value, the kernel performs actions that would otherwise be lost due to an occasional excessive time drift described in previous paragraphs.

4.4 Time Source for Userspace

In HelenOS, userspace tasks don't communicate with the kernel in order to read the system time. Instead, a mechanism that shares kernel time of the the day counters with userspace address spaces is deployed. On the kernel side, during system initialization, HelenOS allocates a frame of physical memory and stores the time of the day counters there. The counters have the following structure:

- first 32-bit counter for seconds,
- 32-bit counter for microseconds and
- second 32-bit counter for seconds.

One of the userspace tasks with capabilities of memory manager (e.g. ns) asks the kernel to map this frame into its address space. Other non-privileged tasks then use IPC to receive read-only share of this memory. Reading time in a userspace task is therefore just a matter of reading memory.

There are two interesting points about this. First, the counters are 32-bit even on 64-bit machines. The goal is to provide subsecond precision with the possibility to span roughly 136 years. Note that a single 64-bit microsecond counter could not be usually read atomically on 32-bit platforms. Unfortunately, on 32-bit platforms it is usually impossible to read atomically two 32-bit counters either. However, a generic protocol is used to guarantee that sequentially read times will create a non-decreasing sequence.

The problematic part is incrementing seconds counter and clearing microseconds counter together once every second. Seconds must be incremented and microseconds must be reset. However, without any synchronization, the two kernel stores and the two userspace reads can arbitrarily interleave. Furthemore, the reader has no chance to detect that the counters were updated only paritally. Therefore three counters are used in HelenOS.

If seconds need to be updated, the kernel increments the first second counter, issues a write memory barrier operation, updates the microsecond counter, issues another write memory barrier operation and increments the second second counter. When only microseconds needs to be updated, no special action is taken by the kernel. On the other hand, the userspace task must always read all three counters in reversed order. A read memory barrier operation must be issued between each two reads. A non-atomic read is detected when the two second counters differ. The userspace library solves this situation by returning higher of them with microseconds set to zero.

Chapter 5

Scheduling

One of the key aims of the operating system is to create and support the impression that several activities are executing contemporarily. This is true for both uniprocessor as well as multiprocessor systems. In the case of multiprocessor systems, the activities are truly happening in parallel. The scheduler helps to materialize this impression by planning threads on as many processors as possible and, when this strategy reaches its limits, by quickly switching among threads executing on a single processor.

5.1 Contexts

The term *context* refers to the set of processor resources that define the current state of the computation or the environment and the kernel understands it in several more or less narrow sences:

- synchronous register context,
- asynchronous register context,
- FPU context and
- memory management context.

The most narrow sense refers to the the synchronous register context. It includes all the preserved registers as defined by the architecture. To highlight some, the program counter and stack pointer take part in the synchronous register context. These registers must be preserved across a procedure call and during synchronous context switches.

The next type of the context understood by the kernel is the asynchronous register context. On an interrupt, the interrupted execution flow's state must be guaranteed to be eventually completely restored. Therefore the interrupt context includes, among other things, the scratch registers as defined by the architecture. As a special optimization and if certain conditions are met, it need not include the architecture's preserved registers. The condition mentioned in the previous sentence is that the low-level assembly language interrupt routines don't modify the preserved registers. The handlers usually call a higher-level C routine. The preserved registers are then saved on the stack by the compiler generated code of the higher-level function. In HelenOS, several architectures can be compiled with this optimization.

Although the kernel does not do any floating point arithmetics[1], it must protect FPU context of userspace threads against destruction by other threads. Moreover, only a fraction of userspace programs use the floating point unit. HelenOS contains a generic framework for switching FPU context only when the switch is forced (i.e. a thread uses a floating point instruction and its FPU context is not loaded in the processor).

The last member of the context family is the memory management context. It includes memory management registers that identify address spaces on hardware level (i.e. ASIDs and page tables pointers).

5.1.1 Synchronous Context Switches

The scheduler, but also other pieces of the kernel, make heavy use of synchronous context switches, because it is a natural vehicle not only for changes in control flow, but also for switching between two

[1] Some architectures (e.g. ia64) inevitably use a fixed set of floating point registers to carry out their normal operations.

kernel stacks. Two functions figure in a synchronous context switch implementation: *`context_save()`* and *`context_restore()`*. Note that these two functions break the natural perception of the linear C code execution flow starting at function's entry point and ending on one of the function's exit points.

When the *`context_save()`* function is called, the synchronous context is saved in a memory structure passed to it. After executing *`context_save()`*, the caller is returned 1 as a return value. The execution of instructions continues as normally until *`context_restore()`* is called. For the caller, it seems like the call never returns[2]. Nevertheless, a synchronous register context, which is saved in a memory structure passed to *`context_restore()`*, is restored, thus transfering the control flow to the place of occurrence of the corresponding call to *`context_save()`*. From the perspective of the caller of the corresponding *`context_save()`*, it looks like a return from *`context_save()`*. However, this time a return value of 0 is returned.

5.2 Threads

A thread is the basic executable entity with some code and a stack. While the code, implemented by a C language function, can be shared by several threads, the stack is always private to each instance of the thread. Each thread belongs to exactly one task through which it shares address space with its sibling threads. Threads that execute purely in the kernel don't have any userspace memory allocated. However, when a thread has ambitions to run in userspace, it must be allocated a userspace stack. The distinction between the purely kernel threads and threads running also in userspace is made by refering to the former group as to kernel threads and to the latter group as to userspace threads. Both kernel and userspace threads are visible to the scheduler and can become a subject of kernel preemption and thread migration anytime when preemption is not disabled.

Thread States In each moment, a thread exists in one of six possible thread states. When the thread is created and first inserted into the scheduler's run queues or when a thread is migrated to a new processor, it is put into the `Entering` state. After some time elapses, the scheduler picks up the thread and starts executing it. A thread being currently executed on a processor is in the `Running` state. From there, the thread has three possibilities. It either runs until it is preemtped, in which case the state changes to `Ready`, goes to the `Sleeping` state by going to sleep or enters the `Exiting` state when it reaches termination. When the thread exits, its kernel structure usually stays in memory, until the thread is detached by another thread using *`thread_detach()`* function. Terminated but undetached threads are in the `Undead` state. When the thread is detached or detaches itself during its life, it is destroyed in the `Exiting` state and the `Undead` state is not reached.

Pseudo Threads HelenOS userspace layer knows even smaller units of execution. Each userspace thread can make use of an arbitrary number of pseudo threads. These pseudo threads have their own synchronous register context, userspace code and stack. They live their own life within the userspace thread and the scheduler does not have any idea about them because they are completely implemented by the userspace library. This implies several things:

- pseudothreads schedule themselves cooperatively within the time slice given to their userspace thread,
- pseudothreads share FPU context of their containing thread and
- all pseudothreads of one userspace thread block when one of them goes to sleep.

5.3 Scheduler

5.3.1 Run Queues

There is an array of several run queues on each processor. The current version of HelenOS uses 16 run queues implemented by 16 doubly linked lists. Each of the run queues is associated with thread priority. The lower the run queue index in the array is, the higher is the priority of threads linked in that run queue and the shorter is the time in which those threads will execute. When kernel code wants to access the run queue, it must first acquire its lock.

[2] Which might be a source of problems with variable liveliness after *`context_restore()`*.

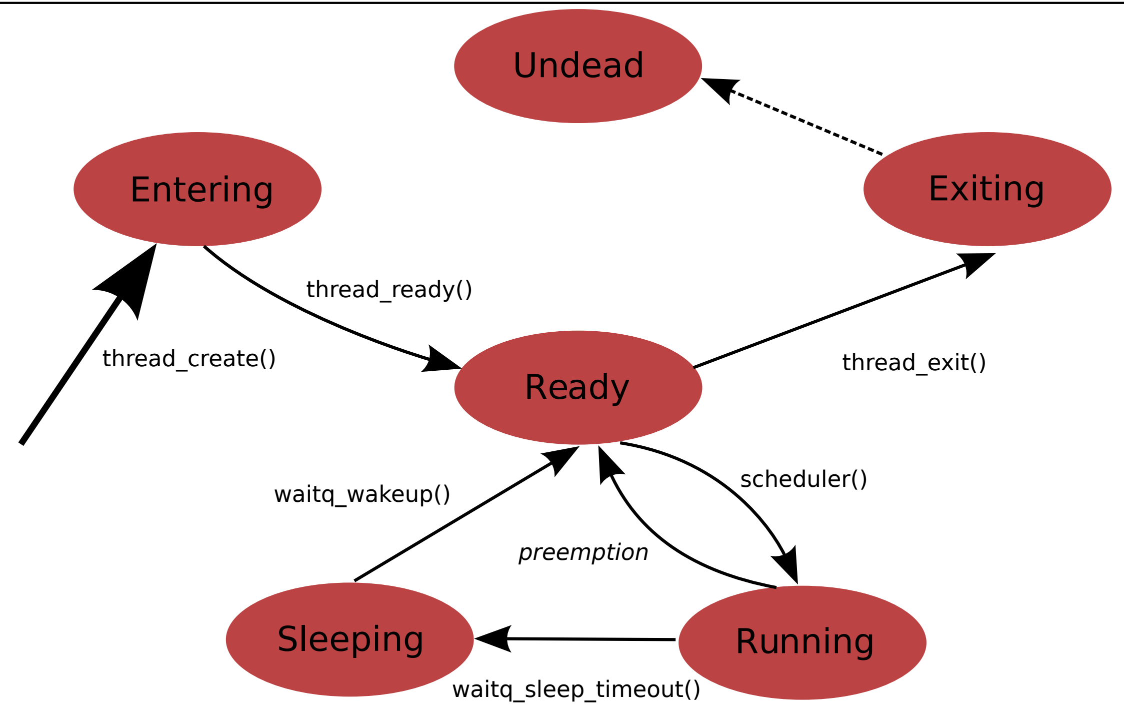

Figure 5.1: Transitions among thread states.

5.3.2 Scheduler Operation

The scheduler is invoked either explicitly when a thread calls the *`scheduler()`* function (e.g. goes to sleep or merely wants to relinquish the processor for a while) or implicitly on a periodic basis when the generic clock interrupt preempts the current thread. After its invocation, the scheduler saves the synchronous register context of the current thread and switches to its private stack. Afterwards, a new thread is selected according to the scheduling policy. If there is no suitable thread, the processor is idle and no thread executes on it. Note that the act of switching to the private scheduler stack is essential. If the processor kept running using the stack of the preempted thread it could damage it because the old thread can be migrated to another processor and scheduled there. In the worst case scenario, two execution flows would be using the same stack.

The scheduling policy is implemented in the function *`find_best_thread()`*. This function walks the processor run queues from lower towards higher indices and looks for a thread. If the visited run queue is empty, it simply searches the next run queue. If it is known in advance that there are no ready threads waiting for execution, *`find_best_thread()`* interruptibly halts the processor or busy waits until some threads arrive. This process repeats until *`find_best_thread()`* succeeds.

After the best thread is chosen, the scheduler switches to the thread's task and memory management context. Finally, the saved synchronous register context is restored and the thread runs. Each scheduled thread is given a time slice depending on its priority (i.e. run queue). The higher priority, the shorter timeslice. To summarize, this policy schedules threads with high priorities more frequently but gives them smaller time slices. On the other hand, lower priority threads are scheduled less frequently, but run for longer periods of time.

When a thread uses its entire time slice, it is preempted and put back into the run queue that immediately follows the previous run queue from which the thread ran. Threads that are woken up from a sleep are put into the biggest priority run queue. Low priority threads are therefore those that don't go to sleep so often and just occupy the processor.

In order to avoid complete starvation of the low priority threads, from time to time, the scheduler will provide them with a bonus of one point priority increase. In other words, the scheduler will now and then move the entire run queues one level up.

5.3.3 Processor Load Balancing

Normally, for the sake of cache locality, threads are scheduled on one of the processors and don't leave it. Nevertheless, a situation in which one processor is heavily overloaded while others sit idle can occur. HelenOS deploys special kernel threads to help mitigate this problem. Each processor is associated with one load balancing thread called `kcpulb` that wakes up regularly to see whether its processor is underbalanced or not. If it is, the thread attempts to migrate threads from other overloaded processors to its own processor's run queues. When the job is done or there is no need for load balancing, the thread goes to sleep.

The balancing threads operate very gently and try to migrate low priority threads first; one `kcpulb` never takes from one processor twice in a row. The load balancing threads as well as threads that were just stolen cannot be migrated. The `kcpulb` threads are wired to their processors and cannot be migrated whatsoever. The ordinary threads are protected only until they are rescheduled.

Chapter 6

Synchronization

6.1 Introduction

The HelenOS operating system is designed to make use of the parallelism offered by the hardware and to exploit concurrency of both the kernel and userspace tasks. This is achieved through multiprocessor support and several levels of multiprogramming such as multitasking, multithreading and also through userspace pseudo threads. However, such a highly concurrent environment needs safe and efficient ways to handle mutual exclusion and synchronization of many execution flows.

6.2 Active Kernel Primitives

6.2.1 Spinlocks

The basic mutual exclusion primitive is the spinlock. The spinlock implements active waiting for the availability of a memory lock (i.e. simple variable) in a multiprocessor-safe manner. This safety is achieved through the use of a specialized, architecture-dependent, atomic test-and-set operation which either locks the spinlock (i.e. sets the variable) or, provided that it is already locked, leaves it unaltered. In any case, the test-and-set operation returns a value, thus signalling either success (i.e. zero return value) or failure (i.e. non-zero value) in acquiring the lock. Note that this makes a fundamental difference between the naive algorithm that doesn't use the atomic operation and the spinlock algortihm. While the naive algorithm is prone to race conditions on SMP configurations and thus is completely SMP-unsafe, the spinlock algorithm eliminates the possibility of race conditions and is suitable for mutual exclusion use.

The semantics of the test-and-set operation is that the spinlock remains unavailable until this operation called on the respective spinlock returns zero. HelenOS builds two functions on top of the test-and-set operation. The first function is the unconditional attempt to acquire the spinlock and is called `spinlock_lock()`. It simply loops until the test-and-set returns a zero value. The other function, `spinlock_trylock()`, is the conditional lock operation and calls the test-and-set only once to find out whether it managed to acquire the spinlock or not. The conditional operation is useful in situations in which an algorithm cannot acquire more spinlocks in the proper order and a deadlock cannot be avoided. In such a case, the algorithm would detect the danger and instead of possibly deadlocking the system it would simply release some spinlocks it already holds and retry the whole operation with the hope that it will succeed next time. The unlock function, `spinlock_unlock()`, is quite easy - it merely clears the spinlock variable.

Nevertheless, there is a special issue related to hardware optimizations that modern processors implement. Particularly problematic is the out-of-order execution of instructions within the critical section protected by a spinlock. The processors are always self-consistent so that they can carry out speculatively executed instructions in the right order with regard to dependencies among those instructions. However, the dependency between instructions inside the critical section and those that implement locking and unlocking of the respective spinlock is not implicit on some processor architectures. As a result, the processor needs to be explicitly told about each occurrence of such a dependency. Therefore, HelenOS adds architecture-specific hooks to all `spinlock_lock()`, `spinlock_trylock()` and `spinlock_unlock()` functions to prevent the instructions inside the critical section from permeating out. On some architectures, these hooks can be void because the dependencies are implicitly there because of the special properties of locking and unlocking instructions. However, other architectures need to instrument these hooks with different memory barriers, depending on what operations could permeate out.

Spinlocks have one significant drawback: when held for longer time periods, they harm both parallelism and concurrency. The processor executing *spinlock_lock()* does not do any fruitful work and is effectively halted until it can grab the lock and proceed. Similarily, other execution flows cannot execute on the processor that holds the spinlock because the kernel disables preemption on that processor when a spinlock is held. The reason behind disabling preemption is priority inversion problem avoidance. For the same reason, threads are strongly discouraged from sleeping when they hold a spinlock.

To summarize, spinlocks represent very simple and essential mutual exclusion primitive for SMP systems. On the other hand, spinlocks scale poorly because of the active loop they are based on. Therefore, spinlocks are used in HelenOS only for short-time mutual exclusion and in cases where the mutual exclusion is required out of thread context. Lastly, spinlocks are used in the construction of passive synchronization primitives.

6.3 Passive Kernel Synchronization

6.3.1 Wait Queues

A wait queue is the basic passive synchronization primitive on which all other passive synchronization primitives are built. Simply put, it allows a thread to sleep until an event associated with the particular wait queue occurs. Multiple threads are notified about incoming events in a first come, first served fashion. Moreover, should the event come before any thread waits for it, it is recorded in the wait queue as a missed wakeup and later forwarded to the first thread that decides to wait in the queue. The inner structures of the wait queue are protected by a spinlock.

The thread that wants to wait for a wait queue event uses the *waitq_sleep_timeout()* function. The algorithm then checks the wait queue's counter of missed wakeups and if there are any missed wakeups, the call returns immediately. The call also returns immediately if only a conditional wait was requested. Otherwise the thread is enqueued in the wait queue's list of sleeping threads and its state is changed to `Sleeping`. It then sleeps until one of the following events happens:

1. another thread calls *waitq_wakeup()* and the thread is the first thread in the wait queue's list of sleeping threads;
2. another thread calls *waitq_interrupt_sleep()* on the sleeping thread;
3. the sleep times out provided that none of the previous occurred within a specified time limit; the limit can be infinity.

All five possibilities (immediate return on success, immediate return on failure, wakeup after sleep, interruption and timeout) are distinguishable by the return value of *waitq_sleep_timeout()*. Being able to interrupt a sleeping thread is essential for externally initiated thread termination. The ability to wait only for a certain amount of time is used, for instance, to passively delay thread execution by several microseconds or even seconds in *thread_sleep()* function. Due to the fact that all other passive kernel synchronization primitives are based on wait queues, they also have the option of being interrutped and, more importantly, can timeout. All of them also implement the conditional operation. Furthemore, this very fundamental interface reaches up to the implementation of futexes - userspace synchronization primitive, which makes it possible for a userspace thread to request a synchronization operation with a timeout or a conditional operation.

From the description above, it should be apparent, that when a sleeping thread is woken by *waitq_wakeup()* or when *waitq_sleep_timeout()* succeeds immediately, the thread can be sure that the event has occurred. The thread need not and should not verify this fact. This approach is called direct hand-off and is characteristic for all passive HelenOS synchronization primitives, with the exception as described below.

6.3.2 Semaphores

The interesting point about wait queues is that the number of missed wakeups is equal to the number of threads that will not block in *watiq_sleep_timeout()* and would immediately succeed instead. On the other hand, semaphores are synchronization primitives that will let predefined amount of threads into their critical section and block any other threads above this count. However, these two cases are exactly the same. Semaphores in HelenOS are therefore implemented as wait queues with a single semantic change: their wait queue is initialized to have so many missed wakeups as is the number of threads that the semphore intends to let into its critical section simultaneously.

In the semaphore language, the wait queue operation *`waitq_sleep_timeout()`* corresponds to semaphore *`down`* operation, represented by the function *`semaphore_down_timeout()`* and by way of similitude the wait queue operation waitq_wakeup corresponds to semaphore *`up`* operation, represented by the function *`sempafore_up()`*. The conditional down operation is called *`semaphore_trydown()`*.

6.3.3 Mutexes

Mutexes are sometimes referred to as binary sempahores. That means that mutexes are like semaphores that allow only one thread in its critical section. Indeed, mutexes in HelenOS are implemented exactly in this way: they are built on top of semaphores. From another point of view, they can be viewed as spinlocks without busy waiting. Their semaphore heritage provides good basics for both conditional operation and operation with timeout. The locking operation is called *`mutex_lock()`*, the conditional locking operation is called *`mutex_trylock()`* and the unlocking operation is called *`mutex_unlock()`*.

6.3.4 Reader/Writer Locks

Reader/writer locks, or rwlocks, are by far the most complicated synchronization primitive within the kernel. The goal of these locks is to improve concurrency of applications, in which threads need to synchronize access to a shared resource, and that access can be partitioned into a read-only mode and a write mode. Reader/writer locks should make it possible for several, possibly many, readers to enter the critical section, provided that no writer is currently in the critical section, or to be in the critical section contemporarily. Writers are allowed to enter the critical section only individually, provided that no reader is in the critical section already. Applications, in which the majority of operations can be done in the read-only mode, can benefit from increased concurrency created by reader/writer locks.

During reader/writer lock construction, a decision should be made whether readers will be prefered over writers or whether writers will be prefered over readers in cases when the lock is not currently held and both a reader and a writer want to gain the lock. Some operating systems prefer one group over the other, creating thus a possibility for starving the unprefered group. In the HelenOS operating system, none of the two groups is prefered. The lock is granted on a first come, first served basis with the additional note that readers are granted the lock in the biggest possible batch.

With this policy and the timeout modes of operation, the direct hand-off becomes much more complicated. For instance, a writer leaving the critical section must wake up all leading readers in the rwlock's wait queue or one leading writer or no-one if no thread is waiting. Similarily, the last reader leaving the critical section must wakeup the sleeping writer if there are any sleeping threads left at all. As another example, if a writer at the beginning of the rwlock's wait queue times out and the lock is held by at least one reader, the writer which has timed out must first wake up all readers that follow him in the queue prior to signalling the timeout itself and giving up.

Due to the issues mentioned in the previous paragraph, the reader/writer lock imlpementation needs to walk the rwlock wait queue's list of sleeping threads directly, in order to find out the type of access that the queueing threads demand. This makes the code difficult to understand and dependent on the internal implementation of the wait queue. Nevertheless, it remains unclear to the authors of HelenOS whether a simpler but equivalently fair solution exists.

The implementation of rwlocks as it has been already put, makes use of one single wait queue for both readers and writers, thus avoiding any possibility of starvation. In fact, rwlocks use a mutex rather than a bare wait queue. This mutex is called *exclusive* and is used to synchronize writers. The writer's lock operation, *`rwlock_write_lock_timeout()`*, simply tries to acquire the exclusive mutex. If it succeeds, the writer is granted the rwlock. However, if the operation fails (e.g. times out), the writer must check for potential readers at the head of the list of sleeping threads associated with the mutex's wait queue and then proceed according to the procedure outlined above.

The exclusive mutex plays an important role in reader synchronization as well. However, a reader doing the reader's lock operation, *`rwlock_read_lock_timeout()`*, may bypass this mutex when it detects that:

1. there are other readers in the critical section and
2. there are no sleeping threads waiting for the exclusive mutex.

If both conditions are true, the reader will bypass the mutex, increment the number of readers in the critical section and then enter the critical section. Note that if there are any sleeping threads at the beginning of the wait queue, the first must be a writer. If the conditions are not fulfilled, the reader normally waits until the exclusive mutex is granted to it.

6.3.5 Condition Variables

Condition variables can be used for waiting until a condition becomes true. In this respect, they are similar to wait queues. But contrary to wait queues, condition variables have different semantics that allows events to be lost when there is no thread waiting for them. In order to support this, condition variables don't use direct hand-off and operate in a way similar to the example below. A thread waiting for the condition becoming true does the following:

```
mutex_lock(mtx);
while (!condition)
        condvar_wait_timeout(cv, mtx); /* the condition is true, do something */
mutex_unlock(mtx);
```

Example 6.1: Use of condvar_wait_timeout().

A thread that causes the condition become true signals this event like this:

```
mutex_lock(mtx);
condition = true;
condvar_signal(cv);  /* condvar_broadcast(cv); */
mutex_unlock(mtx);
```

Example 6.2: Use of condvar_signal().

The wait operation, *condvar_wait_timeout()*, always puts the calling thread to sleep. The thread then sleeps until another thread invokes *condvar_broadcast()* on the same condition variable or until it is woken up by *condvar_signal()*. The *condvar_signal()* operation unblocks the first thread blocking on the condition variable while the *condvar_broadcast()* operation unblocks all threads blocking there. If there are no blocking threads, these two operations have no efect.

Note that the threads must synchronize over a dedicated mutex. To prevent race condition between *condvar_wait_timeout()* and *condvar_signal()* or *condvar_broadcast()*, the mutex is passed to *condvar_wait_timeout()* which then atomically puts the calling thread asleep and unlocks the mutex. When the thread eventually wakes up, *condvar_wait()* regains the mutex and returns.

Also note, that there is no conditional operation for condition variables. Such an operation would make no sence since condition variables are defined to forget events for which there is no waiting thread and because *condvar_wait()* must always go to sleep. The operation with timeout is supported as usually.

In HelenOS, condition variables are based on wait queues. As it is already mentioned above, wait queues remember missed events while condition variables must not do so. This is reasoned by the fact that condition variables are designed for scenarios in which an event might occur very many times without being picked up by any waiting thread. On the other hand, wait queues would remember any event that had not been picked up by a call to *waitq_sleep_timeout()*. Therefore, if wait queues were used directly and without any changes to implement condition variables, the missed_wakeup counter would hurt performance of the implementation: the *while* loop in *condvar_wait_timeout()* would effectively do busy waiting until all missed wakeups were discarded.

The requirement on the wait operation to atomically put the caller to sleep and release the mutex poses an interesting problem on *condvar_wait_timeout()*. More precisely, the thread should sleep in the condvar's wait queue prior to releasing the mutex, but it must not hold the mutex when it is sleeping.

Problems described in the two previous paragraphs are addressed in HelenOS by dividing the *waitq_sleep_timeout()* function into three pieces:

1. `waitq_sleep_prepare()` prepares the thread to go to sleep by, among other things, locking the wait queue;
2. `waitq_sleep_timeout_unsafe()` implements the core blocking logic;
3. `waitq_sleep_finish()` performs cleanup after `waitq_sleep_timeout_unsafe()`; after this call, the wait queue spinlock is guaranteed to be unlocked by the caller

The stock `waitq_sleep_timeout()` is then a mere wrapper that calls these three functions. It is provided for convenience in cases where the caller doesn't require such a low level control. However, the implementation of `condvar_wait_timeout()` does need this finer-grained control because it has to interleave calls to these functions by other actions. It carries its operations out in the following order:

1. calls `waitq_sleep_prepare()` in order to lock the condition variable's wait queue,
2. releases the mutex,
3. clears the counter of missed wakeups,
4. calls `waitq_sleep_timeout_unsafe()`,
5. retakes the mutex,
6. calls `waitq_sleep_finish()`.

6.4 Userspace Synchronization

6.4.1 Futexes

In a multithreaded environment, userspace threads need an efficient way to synchronize. HelenOS borrows an idea from Linux[Franke et al.] to implement lightweight userspace synchronization and mutual exclusion primitive called futex. The key idea behind futexes is that non-contended synchronization is very fast and takes place only in userspace without kernel's intervention. When more threads contend for a futex, only one of them wins; other threads go to sleep via a dedicated syscall.

The userspace part of the futex is a mere integer variable, a counter, that can be atomically incremented or decremented. The kernel part is rather more complicated. For each userspace futex counter, there is a kernel structure describing the futex. This structure contains:

- number of references,
- physical address of the userspace futex counter,
- hash table link and
- a wait queue.

The reference count helps to find out when the futex is no longer needed and can be deallocated. The physical address is used as a key for the global futex hash table. Note that the kernel has to use physical address to identify the futex beacause one futex can be used for synchronization among different address spaces and can have different virtual addresses in each of them. Finally, the kernel futex structure includes a wait queue. The wait queue is used to put threads that didn't win the futex to sleep until the winner wakes one of them up.

A futex should be initialized by setting its userspace counter to one before it is used. When locking the futex via userspace library function `futex_down_timeout()`, the library code atomically decrements the futex counter and tests if it dropped below zero. If it did, then the futex is locked by another thread and the library uses the `SYS_FUTEX_SLEEP` syscall to put the thread asleep. If the counter decreased to 0, then there was no contention and the thread can enter the critical section protected by the futex. When the thread later leaves that critical section, it, using library function `futex_up()`, atomically increments the counter. If the counter value increased to one, then there again was no contention and no action needs to be done. However, if it increased to zero or even a smaller number, then there are sleeping threads waiting for the futex to become available. In that case, the library has to use the `SYS_FUTEX_WAKEUP` syscall to wake one sleeping thread.

So far, futexes are very elegant in that they don't interfere with the kernel when there is no contention for them. Another nice aspect of futexes is that they don't need to be registered anywhere prior to the first kernel intervention.

Both futex related syscalls, `SYS_FUTEX_SLEEP` and `SYS_FUTEX_WAKEUP`, respectivelly, are mere wrappers for ***`waitq_sleep_timeout()`*** and ***`waitq_sleep_wakeup()`***, respectively, with some housekeeping functionality added. Both syscalls need to translate the userspace virtual address of the futex counter to physical address in order to support synchronization accross shared memory. Once the physical address is known, the kernel checks whether the futexes are already known to it by searching the global futex hash table for an item with the physical address of the futex counter as a key. When the search is successful, it returns an address of the kernel futex structure associated with the counter. If the hash table does not contain the key, the kernel creates it and inserts it into the hash table. At the same time, the the current task's B+tree of known futexes is searched in order to find out if the task already uses the futex. If it does, no action is taken. Otherwise the reference count of the futex is incremented, provided that the futex already existed.

Chapter 7

Memory management

In previous chapters, this book described the scheduling subsystem as the creator of the impression that threads execute in parallel. The memory management subsystem, on the other hand, creates the impression that there is enough physical memory for the kernel and that userspace tasks have the entire address space only for themselves.

7.1 Physical memory management

7.1.1 Zones and frames

HelenOS represents continuous areas of physical memory in structures called frame zones (abbreviated as zones). Each zone contains information about the number of allocated and unallocated physical memory frames as well as the physical base address of the zone and number of frames contained in it. A zone also contains an array of frame structures describing each frame of the zone and, in the last, but not the least important, front, each zone is equipped with a buddy system that faciliates effective allocation of power-of-two sized block of frames.

This organization of physical memory provides good preconditions for hot-plugging of more zones. There is also one currently unused zone attribute: `flags`. The attribute could be used to give a special meaning to some zones in the future.

The zones are linked in a doubly-linked list. This might seem a bit ineffective because the zone list is walked everytime a frame is allocated or deallocated. However, this does not represent a significant performance problem as it is expected that the number of zones will be rather low. Moreover, most architectures merge all zones into one.

Every physical memory frame in a zone, is described by a structure that contains number of references and other data used by buddy system.

7.1.2 Frame allocator

The frame allocator satisfies kernel requests to allocate power-of-two sized blocks of physical memory. Because of zonal organization of physical memory, the frame allocator is always working within a context of a particular frame zone. In order to carry out the allocation requests, the frame allocator is tightly integrated with the buddy system belonging to the zone. The frame allocator is also responsible for updating information about the number of free and busy frames in the zone.

Allocation / deallocation Upon allocation request via function `frame_alloc()`, the frame allocator first tries to find a zone that can satisfy the request (i.e. has the required amount of free frames). Once a suitable zone is found, the frame allocator uses the buddy allocator on the zone's buddy system to perform the allocation. During deallocation, which is triggered by a call to `frame_free()`, the frame allocator looks up the respective zone that contains the frame being deallocated. Afterwards, it calls the buddy allocator again, this time to take care of deallocation within the zone's buddy system.

7.1.3 Buddy allocator

In the buddy system, the memory is broken down into power-of-two sized naturally aligned blocks. These blocks are organized in an array of lists, in which the list with index *i* contains all unallocated

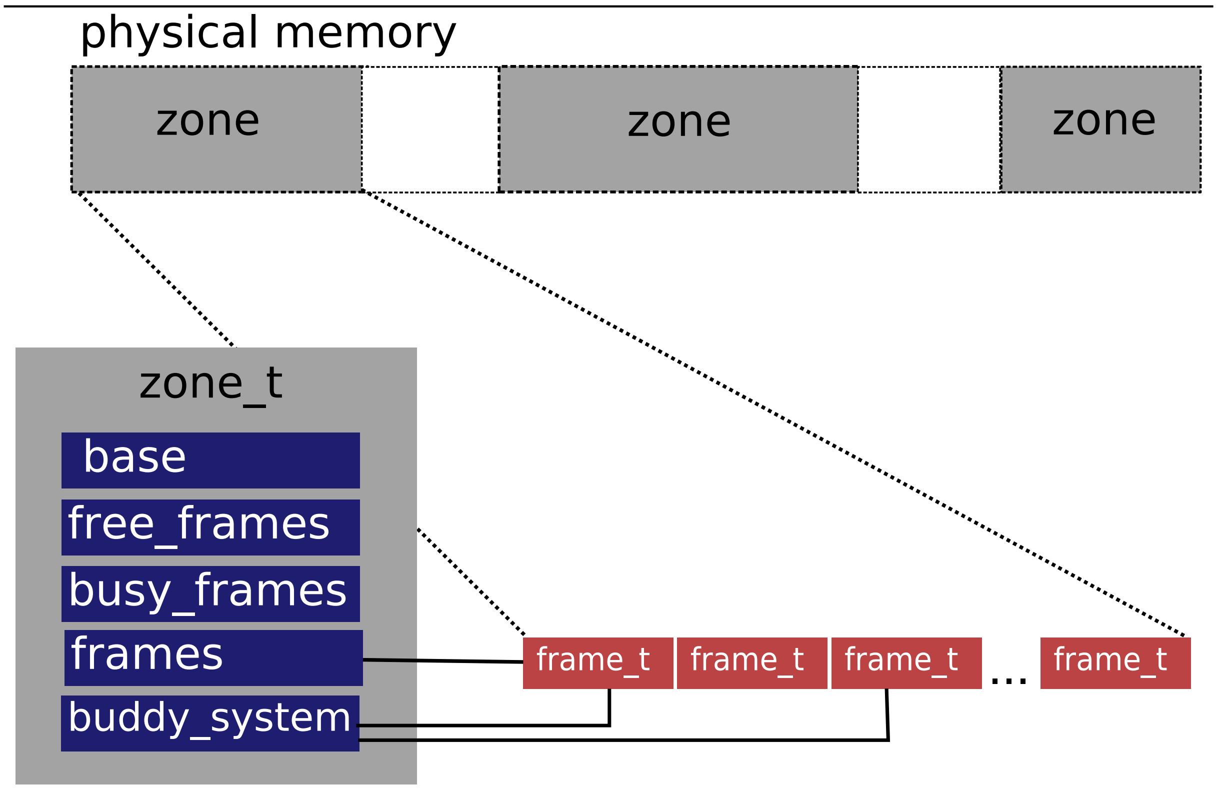

Figure 7.1: Frame allocator scheme.

blocks of size 2^i. The index i is called the order of block. Should there be two adjacent equally sized blocks in the list i (i.e. buddies), the buddy allocator would coalesce them and put the resulting block in list $i + 1$, provided that the resulting block would be naturally aligned. Similarily, when the allocator is asked to allocate a block of size 2^i, it first tries to satisfy the request from the list with index i. If the request cannot be satisfied (i.e. the list i is empty), the buddy allocator will try to allocate and split a larger block from the list with index $i + 1$. Both of these algorithms are recursive. The recursion ends either when there are no blocks to coalesce in the former case or when there are no blocks that can be split in the latter case.

This approach greatly reduces external fragmentation of memory and helps in allocating bigger continuous blocks of memory aligned to their size. On the other hand, the buddy allocator suffers increased internal fragmentation of memory and is not suitable for general kernel allocations. This purpose is better addressed by the slab allocator.

7.1.3.1 Implementation

The buddy allocator is, in fact, an abstract framework which can be easily specialized to serve one particular task. It knows nothing about the nature of memory it helps to allocate. In order to beat the lack of this knowledge, the buddy allocator exports an interface that each of its clients is required to implement. When supplied with an implementation of this interface, the buddy allocator can use specialized external functions to find a buddy for a block, split and coalesce blocks, manipulate block order and mark blocks busy or available.

Data organization Each entity allocable by the buddy allocator is required to contain space for storing block order number and a link variable used to interconnect blocks within the same order.

Whatever entities are allocated by the buddy allocator, the first entity within a block is used to represent the entire block. The first entity keeps the order of the whole block. Other entities within the block are assigned the magic value `BUDDY_INNER_BLOCK`. This is especially important for effective identification of buddies in a one-dimensional array because the entity that represents a potential buddy cannot be associated with `BUDDY_INNER_BLOCK` (i.e. if it is associated with `BUDDY_INNER_BLOCK` then it is not a buddy).

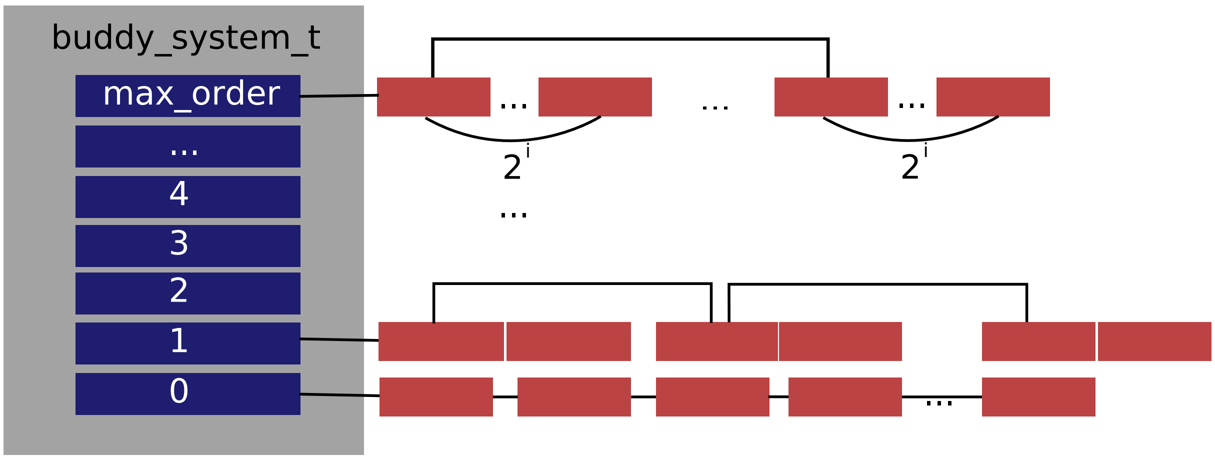

Figure 7.2: Buddy system scheme.

7.1.4 Slab allocator

The majority of memory allocation requests in the kernel is for small, frequently used data structures. The basic idea behind the slab allocator is that commonly used objects are preallocated in continuous areas of physical memory called slabs[1]. Whenever an object is to be allocated, the slab allocator returns the first available item from a suitable slab corresponding to the object type[2]. Due to the fact that the sizes of the requested and allocated object match, the slab allocator significantly reduces internal fragmentation. Slabs of one object type are organized in a structure called slab cache. There are ususally more slabs in the slab cache, depending on previous allocations. If the the slab cache runs out of available slabs, new slabs are allocated. In order to exploit parallelism and to avoid locking of shared spinlocks, slab caches can have variants of processor-private slabs called magazines. On each processor, there is a two-magazine cache. Full magazines that are not part of any per-processor magazine cache are stored in a global list of full magazines. Each object begins its life in a slab. When it is allocated from there, the slab allocator calls a constructor that is registered in the respective slab cache. The constructor initializes and brings the object into a known state. The object is then used by the user. When the user later frees the object, the slab allocator puts it into a processor private magazine cache, from where it can be precedently allocated again. Note that allocations satisfied from a magazine are already initialized by the constructor. When both of the processor cached magazines get full, the allocator will move one of the magazines to the list of full magazines. Similarily, when allocating from an empty processor magazine cache, the kernel will reload only one magazine from the list of full magazines. In other words, the slab allocator tries to keep the processor magazine cache only half-full in order to prevent thrashing when allocations and deallocations interleave on magazine boundaries. The advantage of this setup is that during most of the allocations, no global spinlock needs to be held.

Should HelenOS run short of memory, it would start deallocating objects from magazines, calling slab cache destructor on them and putting them back into slabs. When a slab contanins no allocated object, it is immediately freed.

7.1.4.1 Implementation

The slab allocator is closely modelled after [Bonwick01] with the following exceptions:

- empty slabs are immediately deallocated and
- empty magazines are deallocated when not needed.

The following features are not currently supported but would be easy to do:

- cache coloring and

[1] Slabs are in fact blocks of physical memory frames allocated from the frame allocator.
[2] The mechanism is rather more complicated, see the next paragraph.

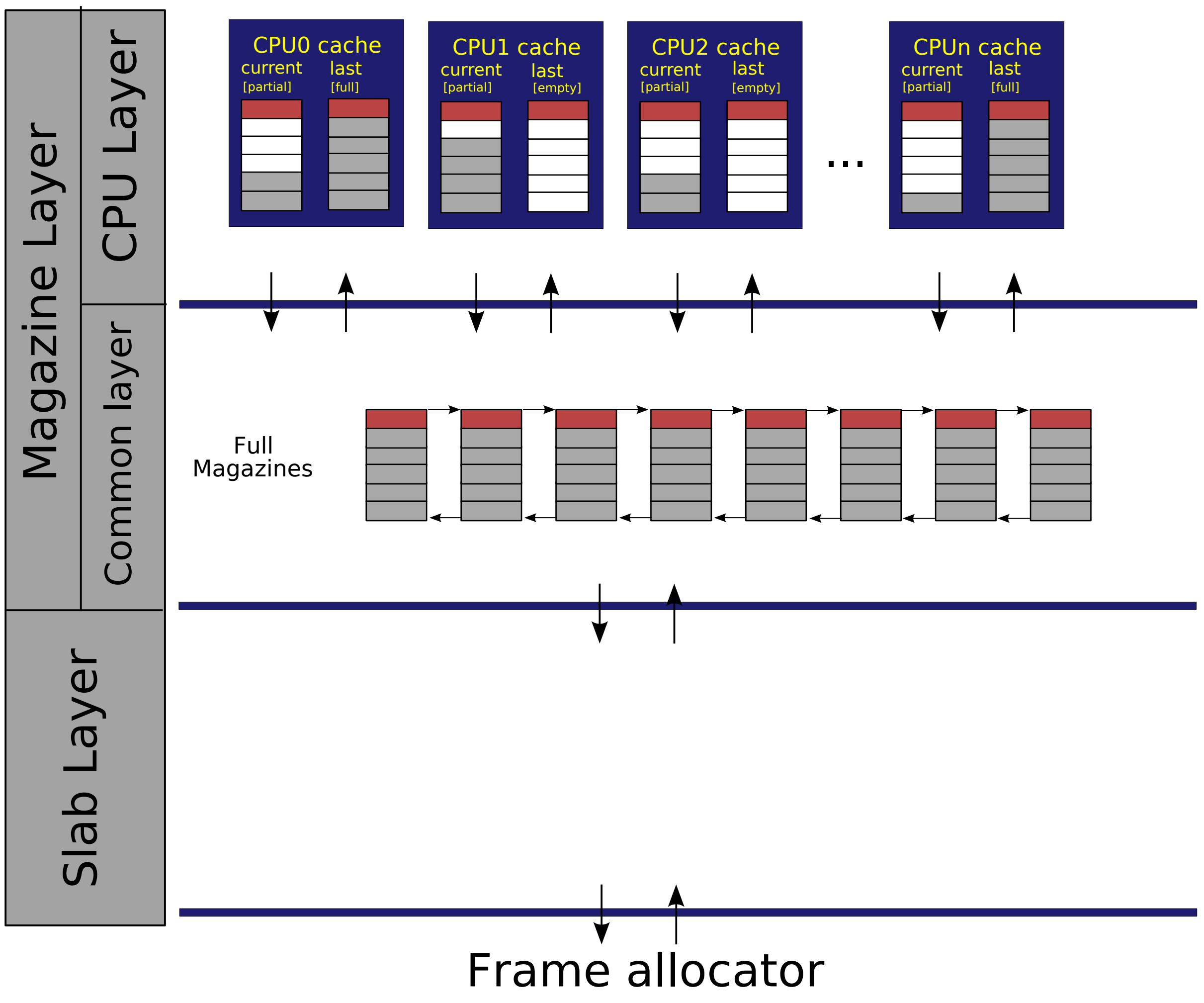

Figure 7.3: Slab allocator scheme.

- dynamic magazine grow (different magazine sizes are already supported, but the allocation strategy would need to be adjusted).

7.1.4.1.1 Allocation/deallocation The following two paragraphs summarize and complete the description of the slab allocator operation (i.e. `slab_alloc()` and `slab_free()` functions).

Allocation *Step 1.* When an allocation request comes, the slab allocator checks availability of memory in the current magazine of the local processor magazine cache. If the available memory is there, the allocator just pops the object from magazine and returns it.

Step 2. If the current magazine in the processor magazine cache is empty, the allocator will attempt to swap it with the last magazine from the cache and return to the first step. If also the last magazine is empty, the algorithm will fall through to Step 3.

Step 3. Now the allocator is in the situation when both magazines in the processor magazine cache are empty. The allocator reloads one magazine from the shared list of full magazines. If the reload is successful (i.e. there are full magazines in the list), the algorithm continues with Step 1.

Step 4. In this fail-safe step, an object is allocated from the conventional slab layer and a pointer to it is returned. If also the last magazine is full,

Deallocation *Step 1.* During a deallocation request, the slab allocator checks if the current magazine of the local processor magazine cache is not full. If it is, the pointer to the objects is just pushed into the magazine and the algorithm returns.

Step 2. If the current magazine is full, the allocator will attempt to swap it with the last magazine from the cache and return to the first step. If also the last magazine is empty, the algorithm will fall through to Step 3.

Step 3. Now the allocator is in the situation when both magazines in the processor magazine cache are full. The allocator tries to allocate a new empty magazine and flush one of the full magazines to the shared list of full magazines. If it is successfull, the algoritm continues with Step 1.

Step 4. In case of low memory condition when the allocation of empty magazine fails, the object is moved directly into slab. In the worst case object deallocation does not need to allocate any additional memory.

7.2 Virtual memory management

Virtual memory is essential for an operating system because it makes several things possible. First, it helps to isolate tasks from each other by encapsulating them in their private address spaces. Second, virtual memory can give tasks the feeling of more memory available than is actually possible. And third, by using virtual memory, there might be multiple copies of the same program, linked to the same addresses, running in the system. There are at least two known mechanisms for implementing virtual memory: segmentation and paging. Even though some processor architectures supported by HelenOS[3] provide both mechanisms, the kernel makes use solely of paging.

7.2.1 VAT subsystem

In a paged virtual memory, the entire virtual address space is divided into small power-of-two sized naturally aligned blocks called pages. The processor implements a translation mechanism, that allows the operating system to manage mappings between set of pages and set of identically sized and identically aligned pieces of physical memory called frames. In a result, references to continuous virtual memory areas don't necessarily need to reference continuos area of physical memory. Supported page sizes usually range from several kilobytes to several megabytes. Each page that takes part in the mapping is associated with certain attributes that further desribe the mapping (e.g. access rights, dirty and accessed bits and present bit).

When the processor accesses a page that is not present (i.e. its present bit is not set), the operating system is notified through a special exception called page fault. It is then up to the operating system to service the page fault. In HelenOS, some page faults are fatal and result in either task termination or, in the worse case, kernel panic[4], while other page faults are used to load memory on demand or to notify the kernel about certain events. The set of all page mappings is stored in a memory structure called page tables. Some architectures have no hardware support for page tables[5] while other processor

[3] ia32 has full-fledged segmentation.
[4] Such a condition would be either caused by a hardware failure or a bug in the kernel.
[5] On mips32, TLB-only model is used and the operating system is responsible for managing software defined page tables.

architectures[6] understand the whole memory format thereof. Despite all the possible differences in page table formats, the HelenOS VAT subsystem[7] unifies the page table operations under one programming interface. For all parts of the kernel, three basic functions are provided:

- *page_mapping_insert()*,
- *page_mapping_find()* and
- *page_mapping_remove()*.

The *page_mapping_insert()* function is used to introduce a mapping for one virtual memory page belonging to a particular address space into the page tables. Once the mapping is in the page tables, it can be searched by *page_mapping_find()* and removed by *page_mapping_remove()*. All of these functions internally select the page table mechanism specific functions that carry out the self operation.

There are currently two supported mechanisms: generic 4-level hierarchical page tables and global page hash table. Both of the mechanisms are generic as they cover several hardware platforms. For instance, the 4-level hierarchical page table mechanism is used by amd64, ia32, mips32 and ppc32, respectively. These architectures have the following page table format: 4-level, 2-level, TLB-only and hardware hash table, respectively. On the other hand, the global page hash table is used on ia64 that can be TLB-only or use a hardware hash table. Although only two mechanisms are currently implemented, other mechanisms (e.g. B+tree) can be easily added.

7.2.1.1 Hierarchical 4-level page tables

Hierarchical 4-level page tables are generalization of the frequently used hierarchical model of page tables. In this mechanism, each address space has its own page tables. To avoid confusion in terminology used by hardware vendors, in HelenOS, the root level page table is called PTL0, the two middle levels are called PTL1 and PTL2, and, finally, the leaf level is called PTL3. All architectures using this mechanism are required to use PTL0 and PTL3. However, the middle levels can be left out, depending on the hardware hierarchy or structure of software-only page tables. The genericity is achieved through a set of macros that define transitions from one level to another. Unused levels are optimised out by the compiler.

7.2.1.2 Global page hash table

Implementation of the global page hash table was encouraged by 64-bit architectures that can have rather sparse address spaces. The hash table contains valid mappings only. Each entry of the hash table contains an address space pointer, virtual memory page number (VPN), physical memory frame number (PFN) and a set of flags. The pair of the address space pointer and the virtual memory page number is used as a key for the hash table. One of the major differences between the global page hash table and hierarchical 4-level page tables is that there is only a single global page hash table in the system while hierarchical page tables exist per address space. Thus, the global page hash table contains information about mappings of all address spaces in the system.

The global page hash table mechanism uses the generic hash table type as described in the chapter dedicated to data structures earlier in this book.

7.3 Translation Lookaside buffer

Due to the extensive overhead of several extra memory accesses during page table lookup that are necessary on every instruction, modern architectures deploy fast assotiative cache of recelntly used page mappings. This cache is called TLB - Translation Lookaside Buffer - and is present on every processor in the system. As it has been already pointed out, TLB is the only page translation mechanism for some architectures.

[6] Like amd64 and ia32.
[7] Virtual Address Translation subsystem.

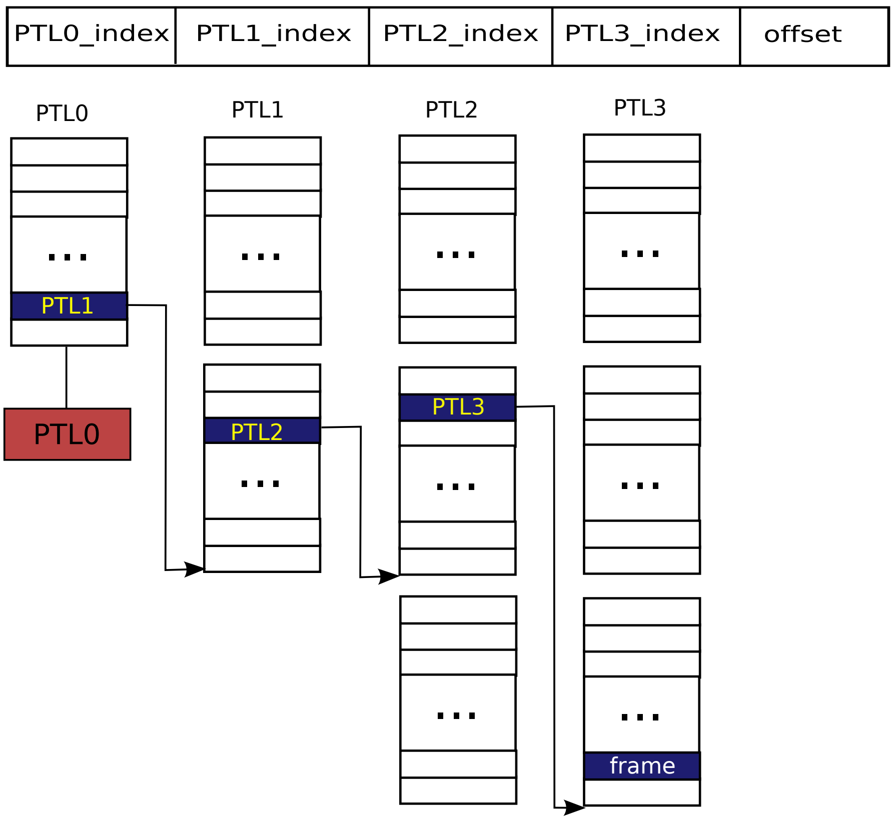

Figure 7.4: Hierarchical 4-level page tables.

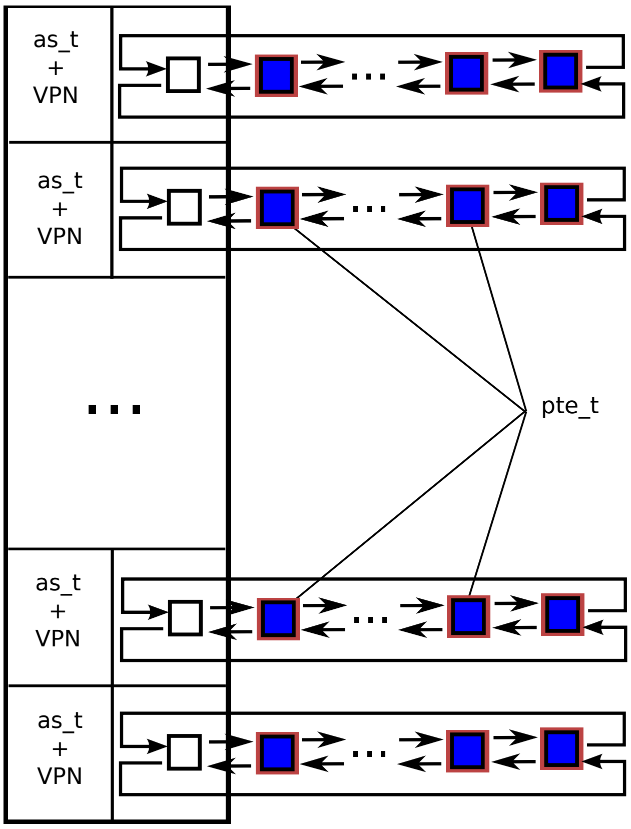

Figure 7.5: Global page hash table.

7.3.1 TLB consistency

The operating system is responsible for keeping TLB consistent with the page tables. Whenever mappings are modified or purged from the page tables, or when an address space identifier is reused, the kernel needs to invalidate the respective contents of TLB. Some TLB types support partial invalidation of their content (e.g. ranges of pages or address spaces) while other types can be invalidated only entirely. The invalidation must be done on all processors for there is one TLB per processor. Maintaining TLB consistency on multiprocessor configurations is not as trivial as it might look from the first glance.

The remote TLB invalidation is called TLB shootdown. HelenOS uses a simplified variant of the algorithm described in [Black89].

TLB shootdown is performed in three phases.

Phase 1. The initiator clears its TLB flag and locks the global TLB spinlock. The request is then enqueued into all other processors' TLB shootdown message queues. When the TLB shootdown message queue is full on any processor, the queue is purged and a single request to invalidate the entire TLB is stored there. Once all the TLB shootdown messages were dispatched, the initiator sends all other processors an interrupt to notify them about the incoming TLB shootdown message. It then spins until all processors accept the interrupt and clear their TLB flags.

Phase 2. Except for the initiator, all other processors are spining on the TLB spinlock. The initiator is now free to modify the page tables and purge its own TLB. The initiator then unlocks the global TLB spinlock and sets its TLB flag.

Phase 3. When the spinlock is unlocked by the initiator, other processors are sequentially granted the spinlock. However, once they manage to lock it, they immediately release it. Each processor invalidates its TLB according to messages found in its TLB shootdown message queue. In the end, each processor sets its TLB flag and resumes its previous operation.

7.4 Address spaces

In HelenOS, address spaces are objects that encapsulate the following items:

- address space identifier,
- page table PTL0 pointer and
- a set of mutually disjunctive address space areas.

Address space identifiers will be discussed later in this section. The address space contains a pointer to PTL0, provided that the architecture uses per address space page tables such as the hierarchical 4-level page tables. The most interesting component is the B+tree of address space areas belonging to the address space.

7.4.1 Address space areas

Because an address space can be composed of heterogenous mappings such as userspace code, data, read-only data and kernel memory, it is further broken down into smaller homogenous units called address space areas. An address space area represents a continuous piece of userspace virtual memory associated with common flags. Kernel memory mappings do not take part in address space areas because they are hardwired either into TLBs or page tables and are thus shared by all address spaces. The flags are a combination of:

- `AS_AREA_READ`,
- `AS_AREA_WRITE`,
- `AS_AREA_EXEC` and
- `AS_AREA_CACHEABLE`.

The `AS_AREA_READ` flag is implicit and cannot be removed. The `AS_AREA_WRITE` flag denotes a writable address space area and the `AS_AREA_EXEC` is used for areas containing code. The combination of `AS_AREA_WRITE` and `AS_AREA_EXEC` is not allowed. Some architectures don't differentiate between executable and non-executable mappings. In that case, the `AS_AREA_EXEC` has no effect on mappings created for the address space area in the page tables. If the flags don't have `AS_AREA_CACHEABLE` set, the page tables content of the area is created with caching disabled. This is useful for address space areas containing memory of some memory mapped device.

Address space areas can be backed by a backend that provides virtual functions for servicing page faults that occur within the address space area, releasing memory allocated by the area and sharing the area. Currently, there are three backends supported by HelenOS: anonymous memory backend, ELF image backend and physical memory backend.

Anonymous memory backend Anonymous memory is memory that has no predefined contents such as userspace stack or heap. Anonymous address space areas are backed by memory allocated from the frame allocator. Areas backed by this backend can be resized as long as they are not shared.

ELF image backend Areas backed by the ELF backend are composed of memory that can be either initialized, partially initialized or completely anonymous. Initialized portions of ELF backend address space areas are those that are entirely physically present in the executable image (e.g. code and initialized data). Anonymous portions are those pages of the *bss* section that exist entirely outside the executable image. Lastly, pages that don't fit into the previous two categories are partially initialized as they are both part of the image and the *bss* section. The initialized portion does not need any memory from the allocator unless it is writable. In that case, pages are duplicated on demand during page fault and memory for the copy is allocated from the frame allocator. The remaining two parts of the ELF always require memory from the frame allocator. Non-shared address space areas backed by the ELF image backend can be resized.

Physical memory backend Physical memory backend is used by the device drivers to access physical memory. No additional memory needs to be allocated on a page fault in this area and when sharing this area. Areas backed by this backend cannot be resized.

7.4.1.1 Memory sharing

Address space areas can be shared provided that their backend supports sharing[8]. When the kernel calls *`as_area_share()`*, a check is made to see whether the area is already being shared. If the area is already shared, it contains a pointer to the share info structure. The pointer is then simply copied into the new address space area and a reference count in the share info structure is incremented. Otherwise a new address space share info structure needs to be created. The backend is then called to duplicate the mapping of pages for which a frame is allocated. The duplicated mapping is stored in the share info structure B+tree called `pagemap`. Note that the reference count of the frames put into the `pagemap` must be incremented in order to avoid a race condition. If the originating address space area had been destroyed before the `pagemap` information made it to the page tables of other address spaces that take part in the sharing, the reference count of the respective frames would have dropped to zero and some of them could have been allocated again.

7.4.1.2 Page faults

When a page fault is encountered in the address space area, the address space page fault handler, *`as_page_fault()`*, invokes the corresponding backend page fault handler to resolve the situation. The backend might either confirm the page fault or perform a remedy. In the non-shared case, depending on the backend, the page fault can be remedied usually by allocating some memory on demand or by looking up the frame for the faulting translation in the ELF image.

Shared address space areas need to consider the `pagemap` B+tree. First they need to make sure whether the mapping is not present in the `pagemap`. If it is there, then the frame reference count is increased and the page fault is resolved. Otherwise the handler proceeds similarily to the non-shared case. If it allocates a physical memory frame, it must increment its reference count and add it to the `pagemap`.

7.4.2 Address Space ID (ASID)

Modern processor architectures optimize TLB utilization by associating TLB entries with address spaces through assigning identification numbers to them. In HelenOS, the term ASID, originally taken from the mips32 terminology, is used to refer to the address space identification number. The advantage of having ASIDs is that TLB does not have to be invalidated on thread context switch as long as ASIDs are

[8] Which is the case for all currently supported backends.

unique. Unfortunately, architectures supported by HelenOS use all different widths of ASID numbers[9] out of which none is sufficient. The amd64 and ia32 architectures cannot make use of ASIDs as their TLB doesn't recognize such an abstraction. Other architectures have support for ASIDs, but for instance ppc32 doesn't make use of them in the current version of HelenOS. The rest of the architectures does use ASIDs. However, even on the ia64 architecture, the minimal supported width of ASID[10] is insufficient to provide a unique integer identifier to all address spaces that might hypothetically coexist in the running system. The situation on mips32 is even worse: the architecture has only 256 unique identifiers. To mitigate the shortage of ASIDs, HelenOS uses the following strategy. When the system initializes, a FIFO queue[11] is created and filled with all available ASIDs. Moreover, every address space remembers the number of processors on which it is active. Address spaces that have a valid ASID and that are not active on any processor are appended to the list of inactive address spaces with valid ASID. When an address space needs to be assigned a valid ASID, it first checks the FIFO queue. If it contains at least one ASID, the ASID is allocated. If the queue is empty, an ASID is simply stolen from the first address space in the list. In that case, the address space that loses the ASID in favor of another address space, is removed from the list. After the new ASID is purged from all TLBs, it can be used by the address space. Note that this approach works due to the fact that the number of ASIDs is greater than the maximal number of processors supported by HelenOS and that there can be only one active address space per processor. In other words, when the FIFO queue is empty, there must be address spaces that are not active on any processor.

[9] amd64 and ia32 don't use similar abstraction at all, mips32 has 8-bit ASIDs and ia64 can have ASIDs between 18 to 24 bits wide.

[10] RID in ia64 terminology.

[11] Note that architecture-specific measures are taken to avoid too large FIFO queue. For instance, seven consecutive ia64 RIDs are grouped to form one HelenOS ASID.

Chapter 8

IPC

Due to the high intertask communication traffic, IPC becomes critical subsystem for microkernels, putting high demands on the speed, latency and reliability of IPC model and implementation. Although theoretically the use of asynchronous messaging system looks promising, it is not often implemented because of a problematic implementation of end user applications. HelenOS implements fully asynchronous messaging system with a special layer providing a user application developer a reasonably synchronous multithreaded environment sufficient to develop complex protocols.

8.1 Kernel Services

Every message consists of four numeric arguments (32-bit and 64-bit on the corresponding platforms), from which the first one is considered a method number on message receipt and a return value on answer receipt. The received message contains identification of the incoming connection, so that the receiving application can distinguish the messages between different senders. Internally the message contains pointer to the originating task and to the source of the communication channel. If the message is forwarded, the originating task identifies the recipient of the answer, the source channel identifies the connection in case of a hangup response.

Every message must be eventually answered. The system keeps track of all messages, so that it can answer them with appropriate error code should one of the connection parties fail unexpectedly. To limit buffering of the messages in the kernel, every task has a limit on the amount of asynchronous messages it can send simultaneously. If the limit is reached, the kernel refuses to send any other message until some active message is answered.

To facilitate kernel-to-user communication, the IPC subsystem provides notification messages. The applications can subscribe to a notification channel and receive messages directed to this channel. Such messages can be freely sent even from interrupt context as they are primarily destined to deliver IRQ events to userspace device drivers. These messages need not be answered, there is no party that could receive such response.

8.1.1 Low Level IPC

The whole IPC subsystem consists of one-way communication channels. Each task has one associated message queue (answerbox). The task can call other tasks and connect its phones to their answerboxes, send and forward messages through these connections and answer received messages. Every sent message is identified by a unique number, so that the response can be later matched against it. The message is sent over the phone to the target answerbox. The server application periodically checks the answerbox and pulls messages from several queues associated with it. After completing the requested action, the server sends a reply back to the answerbox of the originating task. If a need arises, it is possible to *forward* a received message through any of the open phones to another task. This mechanism is used e.g. for opening new connections to services via the naming service.

The answerbox contains four different message queues:

- Incoming call queue
- Dispatched call queue

- Answer queue
- Notification queue

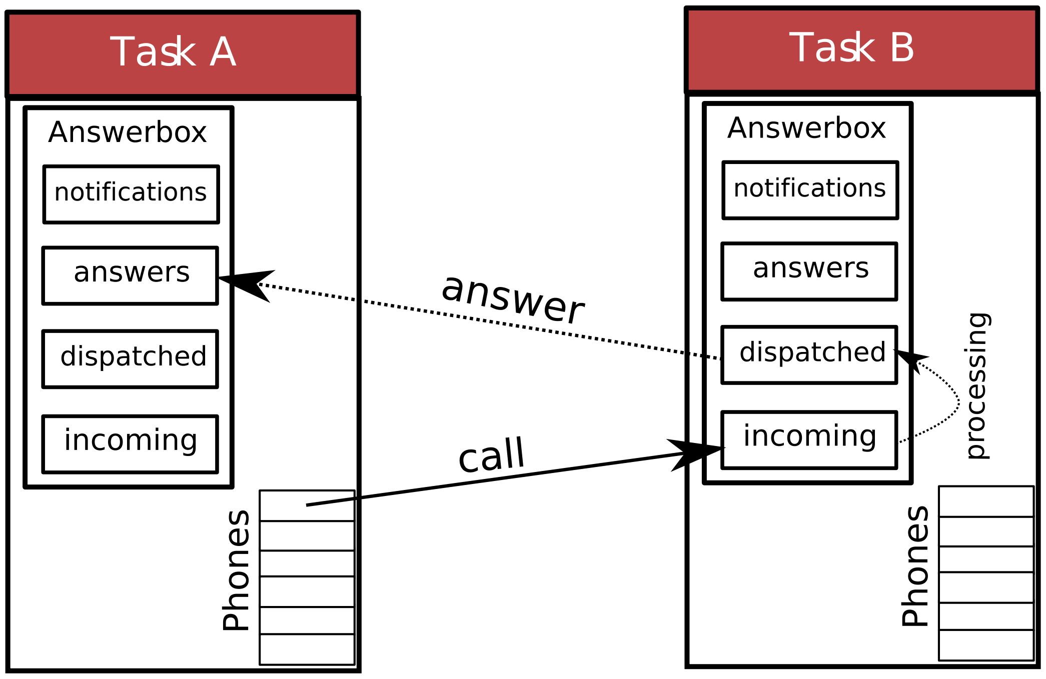

Figure 8.1: Low level IPC

The communication between task A, that is connected to task B looks as follows: task A sends a message over its phone to the target asnwerbox. The message is saved in task B's incoming call queue. When task B fetches the message for processing, it is automatically moved into the dispatched call queue. After the server decides to answer the message, it is removed from dispatched queue and the result is moved into the answer queue of task A.

The arguments contained in the message are completely arbitrary and decided by the user. The low level part of kernel IPC fills in appropriate error codes if there is an error during communication. It is assured that the applications are correctly notified about communication state. If a program closes the outgoing connection, the target answerbox receives a hangup message. The connection identification is not reused until the hangup message is acknowledged and all other pending messages are answered.

Closing an incoming connection is done by responding to any incoming message with an EHANGUP error code. The connection is then immediately closed. The client connection identification (phone id) is not reused, until the client closes its own side of the connection ("hangs his phone up").

When a task dies (whether voluntarily or by being killed), cleanup process is started.

1. hangs up all outgoing connections and sends hangup messages to all target answerboxes,
2. disconnects all incoming connections,
3. disconnects from all notification channels,
4. answers all unanswered messages from answerbox queues with appropriate error code and
5. waits until all outgoing messages are answered and all remaining answerbox queues are empty.

8.1.2 System Call IPC Layer

On top of this simple protocol the kernel provides special services closely related to the inter-process communication. A range of method numbers is allocated and protocol is defined for these functions. These messages are interpreted by the kernel layer and appropriate actions are taken depending on the parameters of the message and the answer.

The kernel provides the following services:

- creating new outgoing connection,
- creating a callback connection,
- sending an address space area and
- asking for an address space area.

On startup, every task is automatically connected to a *naming service task*, which provides a switchboard functionality. In order to open a new outgoing connection, the client sends a `CONNECT_ME_TO` message using any of his phones. If the recepient of this message answers with an accepting answer, a new connection is created. In itself, this mechanism would allow only duplicating existing connection. However, if the message is forwarded, the new connection is made to the final recipient.

In order for a task to be able to forward a message, it must have a phone connected to the destination task. The destination task establishes such connection by sending the `CONNECT_TO_ME` message to the forwarding task. A callback connection is opened afterwards. Every service that wants to receive connections has to ask the naming service to create the callback connection via this mechanism.

Tasks can share their address space areas using IPC messages. The two message types - `AS_AREA_SEND` and `AS_AREA_RECV` are used for sending and receiving an address space area respectively. The shared area can be accessed as soon as the message is acknowledged.

8.2 Userspace View

The conventional design of the asynchronous API seems to produce applications with one event loop and several big switch statements. However, by intensive utilization of userspace pseudo threads, it was possible to create an environment that is not necessarily restricted to this type of event-driven programming and allows for more fluent expression of application programs.

8.2.1 Single Point of Entry

Each task is associated with only one answerbox. If a multithreaded application needs to communicate, it must be not only able to send a message, but it should be able to retrieve the answer as well. If several pseudo threads pull messages from task answerbox, it is a matter of coincidence, which thread receives which message. If a particular thread needs to wait for a message answer, an idle *manager* pseudo thread is found or a new one is created and control is transfered to this manager thread. The manager threads pop messages from the answerbox and put them into appropriate queues of running threads. If a pseudo thread waiting for a message is not running, the control is transferred to it.

Very similar situation arises when a task decides to send a lot of messages and reaches the kernel limit of asynchronous messages. In such situation, two remedies are available - the userspace library can either cache the message locally and resend the message when some answers arrive, or it can block the thread and let it go on only after the message is finally sent to the kernel layer. With one exception, HelenOS uses the second approach - when the kernel responds that the maximum limit of asynchronous messages was reached, the control is transferred to a manager pseudo thread. The manager thread then handles incoming replies and, when space is available, sends the message to the kernel and resumes the application thread execution.

If a kernel notification is received, the servicing procedure is run in the context of the manager pseudo thread. Although it wouldn't be impossible to allow recursive calling, it could potentially lead to an explosion of manager threads. Thus, the kernel notification procedures are not allowed to wait for a message result, they can only answer messages and send new ones without waiting for their results. If the kernel limit for outgoing messages is reached, the data is automatically cached within the application. This behaviour is enforced automatically and the decision making is hidden from the developer.

8.2.2 Ordering Problem

Unfortunately, the real world is is never so simple. E.g. if a server handles incoming requests and as a part of its response sends asynchronous messages, it can be easily preempted and another thread may start intervening. This can happen even if the application utilizes only one userspace thread. Classical synchronization using semaphores is not possible as locking on them would block the thread completely

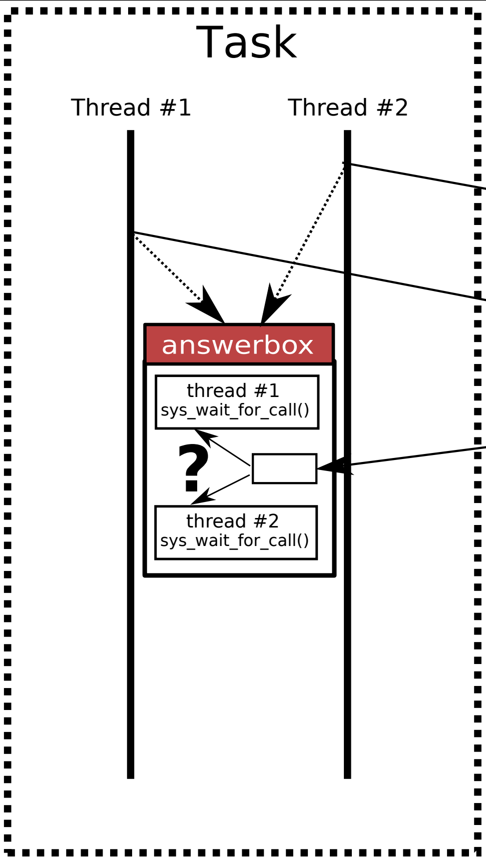

Figure 8.2: Single point of entry

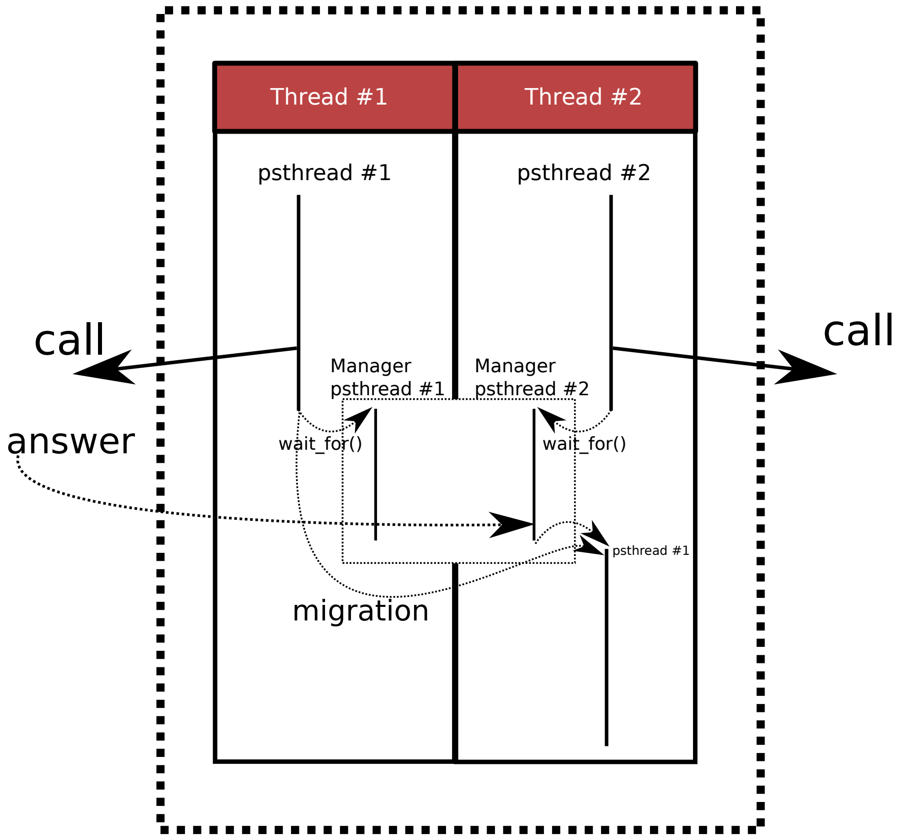

Figure 8.3: Single point of entry solution

so that the answer couldn't be ever processed. The IPC framework allows a developer to specify, that part of the code should not be preempted by any other pseudo thread (except notification handlers) while still being able to queue messages belonging to other pseudo threads and regain control when the answer arrives.

This mechanism works transparently in multithreaded environment, where additional locking mechanism (futexes) should be used. The IPC framework ensures that there will always be enough free userspace threads to handle incoming answers and allow the application to run more pseudo threads inside the usrspace threads without the danger of locking all userspace threads in futexes.

8.2.3 The Interface

The interface was developed to be as simple to use as possible. Typical applications simply send messages and occasionally wait for an answer and check results. If the number of sent messages is higher than the kernel limit, the flow of application is stopped until some answers arrive. On the other hand, server applications are expected to work in a multithreaded environment.

The server interface requires the developer to specify a `connection_thread` function. When new connection is detected, a new pseudo thread is automatically created and control is transferred to this function. The code then decides whether to accept the connection and creates a normal event loop. The userspace IPC library ensures correct switching between several pseudo threads within the kernel environment.

Chapter 9

Device Drivers

Since HelenOS is a microkernel, a framework for supporting userspace device drivers has been implemented. A typical userspace task acting as a device driver might need to:

- receive notifications about interrupts sent by its device,
- access physical memory address space,
- access I/O space and
- control preemption.

9.1 Interrupt Notifications

Userspace tasks that are in hold of the `CAP_IRQ_REG` capability can register themselves via the ***`ipc_register_irq()`*** to be notified about occurrences of a given interrupt. The registration call takes two arguments. The first argument is the IRQ number and the second is the pointer to special pseudocode that instructs the kernel interrupt handler how to process the IRQ. Currently the pseudocode language supports reading and writing physical memory, reading from and writing to I/O space and actions related to running HelenOS in virtual environments.

When the interrupt comes after its handler has been registered by a userspace task, the kernel interrupt handler interprets the pseudocode program and sends an IPC notification to the respective task. The userspace task can get certain information about the interrupt (e.g. what key was pressed) by issuing memory or I/O space reads in the pseudocode program. The read values are wrapped into the IPC notification sent to the task. The write operations are also very essential because some interrupts are level-sensitive and need to be processed in the kernel interrupt routine. In many situations, the interrupt is considered serviced only when the interrupt handler performs certain reads or writes of memory or I/O space.

9.2 Accessing Memory and I/O Space

When a task has the `CAP_MEM_MANAGER` capability, it can use the `SYS_MAP_PHYSMEM` to map regions of physical memory to its address space. When successful, the syscall creates an address space area for the physical memory region. The address space area can be further shared by other tasks. Similarily, when a task has the `CAP_IOSPACE_MANAGER` capability, it is entitled to request access to the I/O space by using the `SYS_IOSPACE_ENABLE`. However, this syscall is relevant only on architectures that have separate I/O space (e.g. amd64 and ia32).

9.3 Disabling Preemption

It might be desirable for a device driver to temporarily disable preemption. Tasks that can do this are required to have the `CAP_PREEMPT_CONTROL` capability. Preemption could be theoretically disabled by disabling interrupts on the current processor, but disabling preemption is more lightweight as interrupt processing remains enabled.

Appendix A

Architecture Specific Notes

A.1 AMD64/Intel EM64T

The amd64 architecture is a 64-bit extension of the older ia32 architecture. Only 64-bit applications are supported. Creating this port was relatively easy, because it shares a lot of common code with ia32 platform. However, the 64-bit extension has some specifics, which made the porting interesting.

A.1.1 Virtual Memory

The amd64 architecture uses standard processor defined 4-level page mapping of 4KB pages. The NX(no-execute) flag on individual pages is fully supported.

A.1.2 TLB-only Paging

All memory on the amd64 architecture is memory mapped, if the kernel needs to access physical memory, a mapping must be created. During boot process the boot loader creates mapping for the first 20MB of physical memory. To correctly initialize the page mapping system, an identity mapping of whole physical memory must be created. However, to create the mapping it is unavoidable to allocate new - possibly unmapped - frames from frame allocator. The ia32 solves it by mapping first 2GB memory during boot process. The same solution on 64-bit platform becomes unfeasible because of the size of the possible address space.

As soon as the exception routines are initialized, a special page fault exception handler is installed which provides a complete view of physical memory until the real page mapping system is initialized. It dynamically changes the page tables to always contain exactly the faulting address. The page then becomes cached in the TLB and on the next page fault the same tables can be utilized to handle another mapping.

A.1.3 Mapping of Physical Memory

The amd64 ABI document describes several modes of program layout. The operating system kernel should be compiled in a *kernel* mode - the kernel is located in the negative 2 gigabytes (0xffffffff80000000-0xffffffffffffffff) and can access data anywhere in the 64-bit space. This wouldn't allow kernel to see directly more than 2GB of physical memory. HelenOS duplicates the virtual mapping of the physical memory starting at 0xffff800000000000 and accesses all external references using this address range.

A.1.4 Thread Local Storage

The code accessing thread local storage uses a segment register FS as a base. The thread local storage is stored in the hidden 64-bit part of the FS register which must be written using priviledged machine specific instructions. Special syscall to change this register is provided to user applications. The TLS address for this platform is expected to point just after the end of the thread local data. The application sometimes need to get a real address of the thread local data in its address space but it is impossible to read the base of the FS segmentation register. The solution is to add the self-reference address to the end of thread local data, so that the application can read the address as %gs:0.

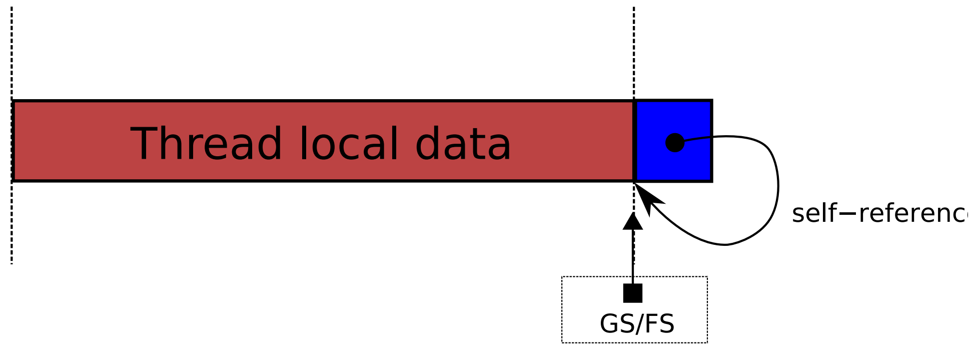

Figure A.1: IA-32 & AMD64 TLD

A.1.5 Fast SYSCALL/SYSRET Support

The entry point for system calls was traditionally a speed problem on the ia32 architecture. The amd64 supports SYSCALL/SYSRET instructions. Upon encountering the SYSCALL instruction, the processor changes privilege mode and transfers control to an address stored in machine specific register. Unlike other similar instructions it does not change stack to a known kernel stack, which must be done by the syscall entry routine. A hidden part of a GS register is provided to support the entry routine with data needed for switching to kernel stack.

A.1.6 Debugging Support

To provide developers tools for finding bugs, hardware breakpoints and watchpoints are supported. The kernel also supports self-debugging - it sets watchpoints on certain data and upon every modification automatically checks whether a correct value was written. It is worthwhile to mention, that since this feature was implemented, the watchpoint was never fired.

A.2 Intel IA-32

The ia32 architecture uses 4K pages and processor supported 2-level page tables. Along with amd64, it is one of the two architectures that fully support SMP configurations. The architecture is mostly similar to amd64, it even shares a lot of code. The debugging support is the same as with amd64. The thread local storage uses GS register.

A.3 32-bit MIPS

Both little and big endian kernels are supported. In order to test different page sizes, the mips32 page size was set to 16K. The mips32 architecture is TLB-only, the kernel simulates 2-level page tables. On processors that support it, lazy FPU context switching is implemented.

A.3.1 Thread Local Storage

The thread local storage support in compilers is a relatively recent phenomena. The standardization of such support for the mips32 platform is very new and even the newest versions of GCC cannot generate 100% correct code. Because of some weird MIPS processor variants, it was decided, that the TLS pointer will be gathered not from some of the free registers, but a special instruction was devised and the kernel is supposed to emulate it. HelenOS expects that the TLS pointer is in the K1 register. Upon encountering the reserved instruction exception and checking that the application is requesting a TLS pointer, it returns the contents of the K1 register. The K1 register is expected to point 0x7000 bytes after the beginning of the thread local data.

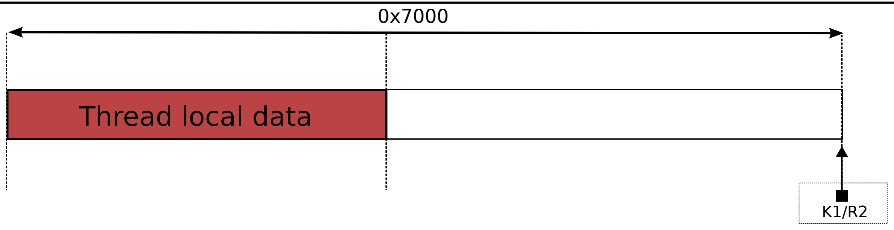

Figure A.2: MIPS & PowerPC TLD

A.3.2 Lazy FPU Context Switching

Implementing lazy FPU switching on MIPS architecture is straightforward. When coprocessor CP1 is disabled, any FPU intruction raises a Coprocessor Unusable exception. The generic lazy FPU context switch is then called that takes care of the correct context save/restore.

A.4 Power PC

PowerPC allows kernel to enable mode, where data and intruction memory reads are not translated through virtual memory mapping (*real mode*). The real mode is automatically enabled when an exception occurs. However, the kernel uses the same memory structure as on other 32-bit platforms - physical memory is mapped into the top 2GB, userspace memory is available in the bottom half of the 32-bit address space.

A.4.1 OpenFirmware Boot

The OpenFirmware loads an image of HelenOS operating system and passes control to the HelenOS specific boot loader. The boot loader then performs following tasks:

- Fetches information from OpenFirmware regarding memory structure, device information etc.
- Switches memory mapping to the real mode.
- Copies the kernel to proper physical address.
- Creates basic memory mapping and switches to the new kernel mapping, in which the kernel can run.
- Passes control to the kernel `main_bsp` function.

A.4.2 Thread Local Storage

The Power PC thread local storage uses R2 register to hold an address, that is 0x7000 bytes after the beginning of the thread local data. Overally it is the same as on the MIPS architecture.

A.5 IA-64

The ia64 kernel uses 16K pages.

A.5.1 Two IA-64 Stacks

The architecture makes use of a pair of stacks. One stack is the ordinary memory stack while the other is a special register stack. This makes the ia64 architecture unique. HelenOS on ia64 solves the problem by allocating two physical memory frames for thread and scheduler stacks. The upper frame is used by the register stack while the first frame is used by the conventional memory stack. The generic kernel and userspace code had to be adjusted to cope with the possibility of allocating more frames for the stack.

A.5.2 Thread Local Storage

Although thread local storage is not officially supported in statically linked binaries, GCC supports it without any major obstacles. The r13 register is used as a thread pointer, the thread local data section starts at address r13+16.

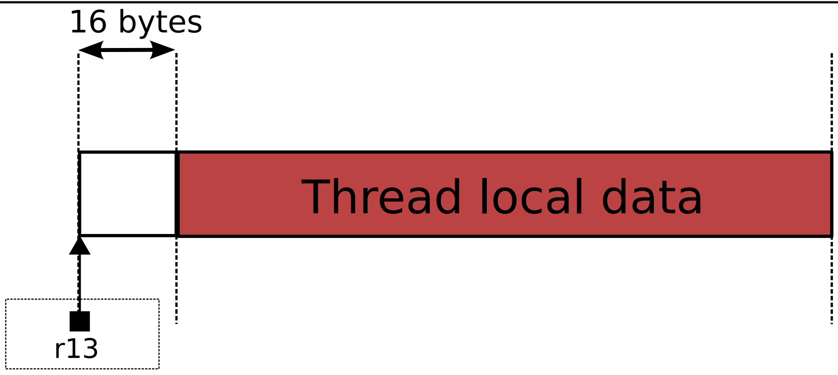

Figure A.3: IA-64 TLD

Appendix B

Bibliography

[Black89]	D.L. Black, R.F. Rashid, D.B. Golub, and C.R. Hill, *Translation Lookaside Buffer Consistency: A Software Approach*, 0163-5964, ACM Press.
[Bonwick01]	Jeff Bonwick and Jonathan Adams, *Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources*, USENIX.
[Franke et al.]	Hubertus Franke, Rusty Russell, and Matthew Kirkwood, *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux*, Proceedings of the 2002 Ottawa Linux Summit.

Index