Evolution of the PikeOS Microkernel

Robert Kaiser
SYSGO AG, Klein-Winternheim, and
Distributed Systems Lab,
University of Applied Sciences, Wiesbaden

Stephan Wagner
SYSGO AG, Klein-Winternheim

E-mail: {rob,swa}@sysgo.com

Abstract

The PikeOS microkernel is targeted at real-time embedded systems. Its main goal is to provide a partitioned environment for multiple operating systems with different design goals to coexist in a single machine. It was initially modelled after the L4 microkernel and has gradually evolved over the years of its application to the real-time, embedded systems space. This paper describes the concepts that were added or removed during this evolution and it provides the rationale behind these design decisions.

1 Introduction

Microkernels have been receiving new attention during the recent years. After being discarded in the mid 1990s on the grounds of causing too much performance impact, the approach seems to be the answer to today's computer problems: Today, computers generally do not suffer from lack of performance, but they often have severe reliability problems. This is especially relevant in the field of embedded systems: While the modern PC user may have (grudgingly) come to accept the occasional system crash as a fact of life, crashing cellular phones or video recorders are embarrassing for their manufacturers and a potential cause for loss of reputation and market share. More critically though, a malfunction in an electronic control unit of, e.g. a car or an airplane can be a severe threat to the life of humans.

Software complexity is the core problem here and microkernels offer the possibility to tackle it with a "divide and conquer" approach: a microkernel only provides basic functionality which can be used to divide the system's resources (memory, I/O devices, CPU time) into separate subsets. Each of these subsets, which we will further refer to as *partitions*, can be regarded as a virtual machine[1] and as such it can host an operating system along with its world of application programs (see Figure 1). Since partitions operate on separate sets of resources, they are completely isolated and there is no way for a program in one partition to affect another partition [2]. In this way, multiple "guest" operating systems are able to coexist in a single machine and their individual functionalities can be tailored to match the requirements of their application programs. Thus, applications are no longer forced to unconditionally trust a huge monolithic kernel containing a lot of complex functionalities that the application may or may not need. Instead, each subsystem can choose the amount of code that it wants to trust: It can trade complexity for trust.

In complex embedded systems, there frequently exist applications with very different temporal requirements: some must guarantee timely service under all circumstances while others have no such constraint, but are instead expected to work "as fast as possible" by making good use of all resources they can possibly get. The differences between such real-time and non-real-time programs are reflected in the functionalities they need their underlying operating system to provide: There are distinct real-time and non-real-time operating system functions. This presents a problem in monolithic systems because there exists only one operating system interface which has to incorporate all the real-time and non-real-time functionalities. In contrast, a microkernel can host multiple operating systems, so it is possible to have distinct real-time or non-real-time interfaces coexisting in separate partitions of a single machine. However, in such a scenario, the microkernel must guarantee timely availability of sufficient computational resources to its real-time guests so they can in turn fulfil their requirements. Not all microkernels are suitable in this respect.

Since the late 1990s, Sysgo have been developing their own microkernel. Initially, it was called "P4" and it was a faithful re-implementation of the L4 version 2.0 microkernel API as specified by Jochen Liedtke in [17]. It was

[1] We consider virtual machine monitors such as Xen [2] or VMware ESX Server [22] to be specialised microkernels.

[2] Unless, of course, both sides explicitly agree on a transaction

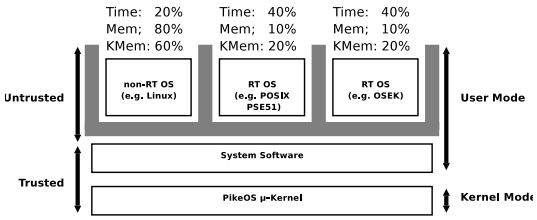

Figure 1. Partitioned system

targeted at embedded systems, therefore, unlike the other then-current L4 implementations, it was written almost entirely in C to facilitate porting, and it was designed to be fully preemptive to support real-time applications.

Over the years of using this kernel in the embedded space, we identified a number of issues with the original L4 version 2.0 interface. These prompted changes of the programming interface and thus, although it is still based on the principles laid out in [16], the PikeOS microkernel's interface today resembles none of the other existing L4 API versions. In the following sections, we will discuss the problems we encountered, and we will outline some of the solutions we chose to apply.

2 Issues

Scheduler Functionality

As already mentioned, there is often a need in embedded systems for programs and operating systems with different temporal requirements to coexist. Real-time programs need to have guaranteed amounts of time at their disposal so they can meet their deadlines. Non-real-time programs need to use all the resources they can get in order to deliver optimal performance. Time allocations for real-time programs must be dimensioned according to worst-case assumptions which are conceivable, but are generally very unlikely to apply. The discrepancy between such a worst case and the average case can easily span multiple orders of magnitude, mainly due to the caches found in every modern computer architecture. Therefore, in the average case, real-time programs usually complete well before their deadlines. In order to put the remaining, excess time to good use, the system should be able to dynamically re-allocate it for use by non-real-time programs.

The classical priority-driven scheduling used by L4 supports this: By assigning sufficiently high priorities to all threads which have temporal requirements, these threads obtain access to the CPU first, and any time not consumed by them is then dynamically made availabe to the lower-priority, non-real-time parts of the system.

However, there is a general problem with this: Any program with a sufficiently high priority has the ability to block all other programs with priorities below its own indefinitely. So, conversely, every program with a given priority is forced to trust in the cooperation of all programs that have a priority higher than its own. There is an implicit dependency between the priority level and the level of trust attributed to a program. But these two attributes do not necessarily coincide: A high priority level may need to be granted to a program because it has a requirement regarding timely execution, whereas a high level of trust should only follow from a thorough inspection of the program code. If a high-priority program exists in one partition and a lower-priority one exists in another, the low priority program must still trust the high-priority one, so this defeats the secure isolation between partitions.

A method is needed here to guarantee (and to enforce) *sufficient* time quanta for partitions hosting real-time programs, thus enabling them to on the one hand provide timely service, while on the other hand keeping them from affecting other partitions by consuming more CPU time than they are entitled to.

Memory Requirements and Memory Accountability

Embedded systems are designed for a specific purpose and they are expected to perform their job at the lowest possible cost. Therefore, an embedded system is usually equipped with just enough memory to do what it is made for, and it is difficult to justify a new concept in the embedded market if it implies a need for more memory.

However, experience using a microkernel like L4 shows that it does increase a system's overall memory requirements. One reason for this is related to the need (or tendency) of using threads to implement certain concepts: Interrupt handlers, for example, are implemented as threads. In L^4Linux, every user process needs *two* L4 threads for its implementation [10]. User-level synchronisation objects (e.g. counting semaphores) are implemented as threads. In result, systems based on L4 tend to employ a rather large number of threads. Each of them requires at least one page of kernel memory for kernel stack and data structures and another page of user memory for the user stack. With a typical load of more than a hundred threads, the resulting memory consumption (8 KB per thread) can be prohibitive for many embedded applications.

Another reason is the kernel's mapping database which is used to store the mapping trees of all of the system's physical pages. It grows and shrinks dynamically as mappings are created or deleted. There is no conceptual limit to the amount of memory that the mapping database might consume. In [18], Liedtke et al describe a problem which has been a topic of ongoing work in the L4 community until today: Malicious (or faulty) programs are able to exhaust

the kernel's memory resources, for example, by requesting a sufficiently large number of mappings to be made. In this way, programs running in one partition can adversely affect programs in other partitions, thereby again defeating the secure isolation between partitions. Furthermore, the mapping database requires a kernel heap allocator[3] to deliver memory blocks which are used to store mapping entries. These entries are made on behalf of different user programs. So, while programs can be charged individually for the kernel memory they consume to store page tables, there is no straightforward way to do this also for the amounts of kernel memory that the mapping database consumes on their behalf.

Code Complexity

The PikeOS kernel is targeted for use in safety-critical applications. Thus it must be prepared for a comprehensive validation according to safety standards such as [20]. Since the kernel runs in privileged mode, all of its code contributes to the trusted code base of every application that might run on top of it. Therefore, the amount of kernel code must be kept minimal. L4 implementations have traditionally been very good in this respect. However, even in L4 there is a kernel mapping database which, besides its problematic consumption of kernel memory, is also a complex piece of code, and so is the underlying slab allocator. This makes it hard (read: costly) to validate.

On the other hand, some L4 concepts tend to force unnecessary complexity on user level code. An example for this would be the construction of address spaces by means of IPC: The creator of an address space has to provide a client thread to run in the new address space for the sole purpose of accepting its mappings. Creation of an address space is an operation that all user-level programs (including safety-critical ones) need to do at some point, so it does not help to reduce kernel complexity at the cost of making these operations overly complex at the user level: either way, the complex code will be part of the trusted code base.

Access Privileges

Application programs running under a guest operating system need not (and, according to the principle of least privilege, should not) be able to access the microkernel's system call interface directly. In fact, they should not even be aware of the microkernel's presence. Otherwise, for example, a Linux user process could change its address space layout without the Linux kernel knowing about it. However, the L4 version 2.0 interface does not provide any means to restrict access to its system call interface. Thus, any thread is able to consume kernel resources (e.g. by installing mappings), to manipulate system settings or to delay a given schedule, etc.

Furthermore, L4 lacks a flexible but powerful IPC control mechanism to manage the information flow of a complex system easily and effectively. The "clans and chiefs" method introduced in [15] is generally regarded to be a simple, but too inflexible solution.

[3]The PikeOS kernel originally employed a "slab" allocator ([4]) for this purpose.

3 Approaches

The scope of this paper does not allow for an exhaustive discussion of the details of all the changes that were made. Therefore, we will concentrate mainly on the two changes that will probably be considered the most radical ones by the L4 community, namely the addition of partition scheduling and the removal of the mapping database. Subsequently, we will briefly discuss some more modifications, ordered by relevance.

Partition Scheduling

The goal of PikeOS is to provide partitions (or virtual machines) that comprise a subset of the system's resources. Processing time is one of those resources. We expect the partitions to host a variety of guest operating systems with different requirements regarding timely execution. There will typically be real-time as well as non-real-time systems, and the real-time systems will generally fall into one of two categories:

- Time-triggered: Threads are executed according to a static schedule which activates each thread at predetermined points in time and for a predetermined amount of time.
- Event-triggered: Threads are activated in response to external events. Scheduling priorities are used to decide which thread is activated first in case multiple events occur at the same time.

Both approaches have their specific advantages and disadvantages[7]. They are mutually exclusive: Allowing for events to interrupt a time-triggered schedule affects its determinism (it increases jitter). On the other hand, reserving a time slot in the schedule for processing any pending events leads to poor worst case response times and –again– jitter of event-triggered threads. So, whenever the two approaches are combined in a system, one of them has to be given precedence and as a consequence, the other is destined to perform poorly.

We can not expect guest operating systems to trust each other. Therefore, a scheduling technique had to be devised

that on one hand allows real-time and non-real-time systems to coexist efficiently, but that on the other hand avoids the implicit dependency between priority and trust we described earlier. This method had to be efficient and simple in order to not add to the complexity of trusted code. To our knowledge, the scheduling method used in the PikeOS kernel is both unique and new ([13]). It is a superset to the method described in the ARINC 653 standard [1], which is commonly applied in the field of avionics:

Like L4, the PikeOS microkernel uses threads with static priority levels to represent activities. But unlike L4, they are grouped into sets which we refer to as "time partitions", τ_i. The microkernel supports a configurable number of such time partitions. Each of them is represented in the microkernel as an array of linked lists (one list per priority level). Threads that have the same priority level are linked into the same list in a first in/first out manner, and the thread at the head of the list is the first to be executed. When a thread blocks, it is appended to the end of the list. So, within each time partition, there is the same scheduling method that L4 uses, i.e. a classical, priority-driven scheduling with round robin scheduling between threads at the same priority. However, unlike L4, the PikeOS microkernel supports multiple time partitions instead of just one. Threads can only execute while their corresponding time partition is active, regardless of their priority. If we cycled through the time partitions, activating each one at a time for a fixed duration, we would obtain the behaviour of an ARINC 653 scheduler. But in contrast to this, the PikeOS kernel allows *two* of the time partitions to be active at the same time:

- The first time partition, τ_0 plays a special role in that it is always active. It is referred to as the "background" partition.
- Of all other partitions $\tau_i (i \neq 0)$, only one can be active at a time. The microkernel provides a (privileged) system call to select the currently active time partition. It is referred to as the "foreground" partition. Switching happens cyclically, according to a pre-configured, static schedule.

The microkernel scheduler always selects for execution the thread with the highest priority from the set union of τ_0 and τ_i. Figure 2 shows the principle.

The threads have different semantics, depending on their priorities and on their time partitions:

- $\tau_i (i \neq 0)$: The system cyclically activates each of the possible τ_i in turn, giving each of them a configurable portion of the cycle time. So, the totality of all threads in any of these time partitions receives a fixed amount of time at fixed points in time. During their active time slice, the threads compete for the CPU according to their priorities. Generally, these threads will be configured to use a mid-level range of priorities (though this is not technically necessary).
- τ_0: The semantics of the threads assigned to the background partition, τ_0, depend on their respective priorities.
  - Low-priority threads within τ_0 will receive the processing time that was assigned to, but not used by the mid-priority threads in τ_i. All threads that do not have any real-time requirements are therefore assigned to τ_0, and they are all given the same, low priority level. Since their priorities are equal, they run under a round robin scheduler, sharing their amount of computation time evenly.
  - Mid- or high-priority threads in τ_0 compete with the currently active foreground partition. If their priorities are higher, they can preempt any low- or mid-priority threads at any time. This is used to implement event-driven real-time threads.

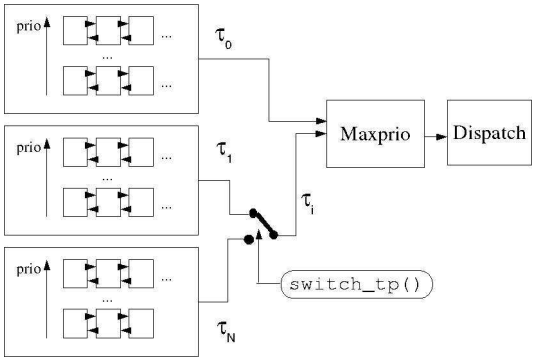

Figure 2. PikeOS partition scheduler: principle of operation.

The relation between the priorities of the foreground and the background partition decides wether time-driven threads take precedence over event-driven ones or vice versa: If an event-driven thread's priority is higher, it can preempt time-driven threads at any time, effectively "stealing" its execution time from them. Therefore, the points in time when time-driven threads are activated become undeterministic to some extent (i.e. they "jitter"). The maximum jitter is increased by the cumulative worst case execution times of all high-priority threads (which has to be bounded). It should be noted that in this case, the high-priority threads have to be trusted to not exceed their execution time.

In the opposite case, i.e. the time-triggered threads having a higher priority, the situation is reversed: the event-triggered threads need to trust in the time-triggered threads

to leave over sufficient time for processing events. This can simply be done by leaving a part of the schedule unallocated.

The PikeOS partition scheduler can not completely solve the dilemma between time-triggered and event-triggered real-time systems, but it does offer good flexibility in this respect: The decision wether to give precedence to time-triggered or event-triggered threads can be made individually for every time-triggered partition, by selecting the priorities of its threads to be either above or below those of the event-triggered threads.

The microkernel implements only the mechanism[4] to select one of the time partitions as active foreground partition. The policy part of deciding which of the time partitions is activated when is left to the user level. This can be done by an interrupt handler thread which runs at a high priority in τ_0. In a typical configuration, this thread gets activated by an external one-shot timer whenever the timeslice of the currently active foreground partition has expired. It then activates the next time partition, re-programs the one-shot timer to trigger an interrupt when the next partition's time allocation expires and then waits for this interrupt. However, this is only one possible scenario: Since it is implemented at the user level, the partition switching policy can easily be replaced without any changes to the microkernel.

In practice, for a safe coexistence of multiple real-time systems, each of them should have its own time slot, i.e. its threads should be assigned to one of the foreground partitions $\tau_i (i \neq 0)$. In this way, each of them can trust to receive a guaranteed amount of time at cyclically repeated (i.e. fixed) points in time. This is a necessary requirement when applying scheduling analysis to a real-time subsystem (see, for example [3]). The possibility to override this strict time-driven scheduling scheme by high-priority background partition threads is reserved for selected activities which can be trusted to consume only negligible amounts of CPU time [5]. With all real-time systems being confined to their individual time slots, their worst case response time and jitter directly depend on the cycle frequency at which their time slots are repeated (see [12]). It would be desirable to make this frequency as high as possible but this is limited by the cost of switching between time partitions. Therefore, the scheduling algorithm that is executed as part of these switches has to be to be both fast (i.e. non-complex) and bounded. The PikeOS partition scheduler meets these requirements: since it has to consider only two partitions for each switch, its execution time is constant (i.e. O(1)). Practical experience with a PowerPC platform[6] has shown worst case partition switch times to be in the range of 25μs. Therefore, if an application can live –for example– with a context switch overhead of 10%, minimum per partition time allocations can be made as low as 250μs.

Recently, a new scheduling method called "adaptive partitioning" [5] has been announced by QNX software systems. Like the PikeOS scheduler, this approach pursues the goal of combining deterministic time allocation with good processor utilisation. It also represents groups of threads as partitions and uses a combination of priority-driven and time-driven scheduling, but no per-partition time slots are defined, i.e. all partitions can compete for the CPU at all times, based on their thread's priorities. This would suggest an O(n) complexity of the scheduler, however sufficient details about the implementation are not available. The guaranteed time allocations of partitions are specified as relative values (percentages of the total CPU processing capacity), but it is not clear how these could be mapped to time slots, i.e. points in time when a partition receives the CPU and durations for how long it will be able to retain it. The approach uses an "averaging window" [7] during which a partition is entitled to receive its percentage of CPU capacity, but it also allows special "critical partitions" to exceed their allocations during an averaging window, as long as they later repay the "debt" which they accumulate in this way. In summary, this approach seems to be more complex than the PikeOS partition scheduler, which would imply a more complex trusted code base. It appears to be targeted mainly towards soft real-time applications: while it is able to guarantee a deterministic average distribution of CPU resources to its partitions, the exact time slot during which a particular partition has access to the CPU is not very well defined.

Abandonment of the Mapping Database

The method of creating address spaces recursively on top of other address spaces as described by Liedtke in [16] is an intriguingly elegant mechanism. However, the concept calls for a recursive unmap operation which in turn necessitates a mapping database to keep track of the mapping history of all pages. This mapping database is the source of many problems, most of which have already been introduced (i.e. potentially unlimited kernel memory consumption, difficulty to attribute the memory to individual kernel resource partitions, general code complexity). An additional problem which is relevant especially for real-time systems is the difficulty in estimating the worst case execution time needed for the recursive termination of task trees. The sum of all these problems provided a strong motivation to question the need for a mapping database in general, and an analysis of the practical use cases in the context of PikeOS revealed that recursive unmap operations normally do not occur:

[4] Called `switch_tp()` in Figure 2

[5] The microkernel maintains a "maximum controlled priority" as defined in the L4 version 2.0 API [17] to ensure that untrusted activities can not assume priorities above their limit.

[6] Motorola MPC5200 at 400 MHz.

[7] The size of this window, according to [5], is typically in the range of 100 milliseconds.

The address spaces in which guest operating systems exist are constructed and destroyed by so-called ”head tasks” (see figure 3). The only time when these head tasks unmap pages is when they destroy their child's address space. In this special case, recursive unmapping is not needed since all address spaces that have been constructed on top of the one being destroyed are also destroyed anyway. Technically, a head task could revoke mappings without destroying the child's address space, but the PikeOS head tasks can be trusted to not do this. Since they are the parents of all user level processes and since a task must fully trust its parent anyway, this is not a new requirement.

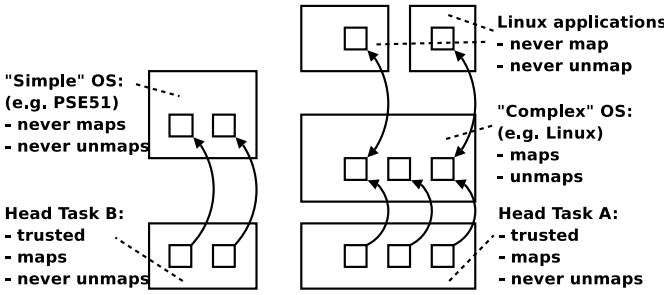

Figure 3. Simplified mappings in PikeOS

The range of guest operating systems that PikeOS can host in its partitions goes from relatively ”simple”, typically real-time and/or safety-critical systems, up to ”complex” operating systems with virtual memory support (e.g. Linux). The former usually exist in a single, static address space. They do not require any mapping functionality themselves. In fact, these subsystems could do just as well with memory protection only. The latter implement their own memory management. These create tasks to whom they map pages, but those tasks do not create any subtasks themselves[8] to whom they could map pages. Thus, all requests to revoke a mapping are either made from within a task itself or from its direct parent. These non-recursive unmap operations, however, can be implemented without a mapping database. Therefore, it was possible to remove it without substitution.

Recent research has shown formal verification of an L4 mapping database to be feasible ([23]). Thus, the code complexity argument made above against the mapping database may have lost some of its weight in the meantime. Nevertheless, the fact remains that validation/verification of a mapping database is a major task and, since its functionality is not required in the context of the PikeOS system, its removal still remains justified.

[8]This is usually enforced by not giving the tasks the ”ability” (see below) to access the microkernel interface. And even without such an enforcement, the effects of then-possible violations would still remain confined to their partitions.

There exists at least one other L4 variant which, according to [19], also works without a mapping database [21]. Like PikeOS, this L4 variant explicitly addresses real-time, embedded systems, which suggests a similar motivation for this modification (although it has not been discussed in any paper we are aware of).

Kernel Resource Partitioning

To facilitate the accounting of kernel resources and thereby to avoid the possibility of denial-of-service attacks as described in [18], the concept of *kernel resource partitions* was introduced into the PikeOS microkernel. The approach is straightforward: The global kernel memory pool of L4 was split into a configurable subset of pools, which we refer to as *kernel resource partitions*. Every task is assigned to one of these partitions at creation time. Whenever the kernel allocates memory on behalf of a user thread, that memory is taken from the corresponding kernel resource partition. If the memory resources of a partition are exhausted, no further allocation for this partition is possible. Other resource partitions, however, are not affected. Thus, denial-of-service attacks are only possible among threads which belong to the same resource partition. The configuration of the resource partitions is done by a specific system call. The initial task, σ_0, has the ”ability” (see below) to make this call. Since all other tasks are ultimately children of σ_0, and since σ_0 can be trusted to not pass this ability on to any of them, it is effectively the only task that is able to use this system call.

Clearly, this approach is a simplistic solution to the problem of kernel resource allocation: it is quite inflexible since all kernel memory resources have to be statically defined at boot time. Nevertheless, for the PikeOS system, it serves the purpose of enabling safe isolation between partitions while keeping the trusted code base small. Its inflexibility is not as big a problem in practice as it may seem: For real-time systems, it is usually not difficult to determine their worst-case kernel memory requirements beforehand and – unlike their time requirements– the discrepancy between worst case and average case is generally not as big. For general-purpose systems like Linux, kernel memory consumption is definitely not easy to predict. However, these systems can implement strategies for graceful degradation: The PikeOS Linux server, for example, flushes its client's mappings when its kernel resource partition runs out of memory, thus freeing up the kernel memory used for their page tables. This leads to a performance degradation akin to TLB thrashing, but, at least, the system remains stable.

The problem of managing kernel memory requirements has been a topic of active research in the L4 community for several years, and it still is. Several approaches have been proposed (e.g. [8, 6], but apparently, the final solution is

yet to be determined. These approaches aim to solve the far more challenging problem of dynamically changing kernel memory requirements, while PikeOS can live with a static, per partition allocation.

Mapping system call

To enable fast and simple creation of entire address spaces, the PikeOS kernel provides a special mapping system call. This system call is restricted to either the caller's address space, or to its child's address space. The receiver of such a mapping does not have to explicitly agree to receive the mapping. At first glance, this looks like a violation of Liedtke's principle that any transactions between address spaces must be explicitly agreed upon by all parties [16]. However, since the source of the mapping in this case is either the task itself (i.e. a trusted party), or its parent, which could just as well create a thread in the target address space under its own control that would then accept the mappings, this is not really a security risk. Although not strictly necessary, this facility greatly simplifies the process of constructing address spaces at the user-level side, while the added complexity at the kernel side is almost negligible.

Events

Many of the services that where implemented on top of the PikeOS microkernel have a requirement for a basic, asynchronous communication primitive (e.g. a counting semaphore). The L4 API only supports synchronous IPC. A counting semaphore can easily be implemented in user space based on the IPC call, so, according to the principle of minimality, this is how it should be done. However, this would require a user thread managing the semaphore object. The resulting memory overhead[9] was considered high enough to justify a slight violation of the minimality principle by introducing a new concept, the *event*, into the kernel interface.

Events are counting semaphores associated to threads (i.e. every thread implicitly has one). A thread can wait on its event by making an appropriate system call. This decrements the event counter and it blocks the caller if the resulting counter is less than zero. The counter is incremented by other threads signalling events to the thread. If the counter is incremented to zero, its associated thread is unblocked again. The rules for sending events are similar to those for IPC, i.e. the recipient must explicitly allow any potential sender threads to signal events.

[9] I.e. 8 kB for a thread that implements a counter – a 32-bit object!

Access Privileges

The PikeOS kernel provides functionality to restrict access to the microkernel's system call interface on a per task basis. To implement this, the concept of *abilities* was added to the kernel: Each ability enables access to a set of system calls. The kernel checks a task's abilities with each system call. Depending on the settings, attempting a system call without sufficient abilities either results in an error, or an exception message being sent to the caller's exception handler. The latter can be used to virtualise system calls.
Abilities are assigned to a task during its creation by the creating *parent* task. They are thus a task property and are stored in the corresponding task descriptor data structure which is maintained by the microkernel. They can not be changed during the lifetime of the task. The kernel ensures that a parent can not give its child any abilities that it does not have itself, i.e. a parent can further restrict its child's abilities, but it can not extend them. It is possible to either disallow certain groups of system calls based on their specific functionality (like system-calls manipulating threads, etc.), or to disallow any system calls at all.
The concept of abilities –although developed independently– is similar to the fine-grained kernel access privileges of MINIX 3 ([9]). The L4::Pistachio kernel variant supports the notion of "privileged threads", which are entitled (by virtue of being members of an elite group of tasks) to make certain system calls. This also bears some similarity with PikeOS's abilities, however the selection of accessible system calls is pre-set and the right to access them can not be passed to tasks outside of the elite group.
To control IPC communication, the PikeOS kernel assigns *communication rights* to each task. A thread is only allowed to send an IPC or to signal an event to another thread if its task has the right of communication with the destination task. By default, every task is only allowed to communicate with its parent, but a parent is able to both grant the communication right between any of its children (i.e. siblings) and to grant communication rights with arbitrary, unrelated tasks.

4 Conclusion

The PikeOS microkernel is an early spin-off from the line of L4 kernel development, which has been adapted to the specific needs of safety-critical real-time embedded systems. It features partitioning of both temporal as well as spatial resources. Compared to other L4 variants that have been developed in parallel, it focuses on real-time computing and minimising trusted code, sacrificing flexibility in some cases.

It currently runs on various PowerPC, MIPS and ia-32

machines. A number of different real-time as well as non real-time operating system interfaces have been ported to run in the microkernel's partitions. Among them are Linux, POSIX PSE51, ITRON, OSEK OS, an Ada runtime system, a JVM and a Soft-PLC runtime system. Some performance comparisons between the PikeOS Linux server and native Linux on x86 platforms have been published in [14]. The average performance impact of 22% is slightly higher than that of other microkernel-based Linux implementations, however, optimizing the performance of Linux has always been a secondary concern for this system.

5 Outlook

Current development of PikeOS goes in several directions:

- Single threaded kernel: The PikeOS microkernel –like Fiasco ([11])– was designed to be fully preemptive. However, a fully preemptive kernel is always more complex, opening up many possibilities for, e.g., race conditions which are hard to detect. It also requires more resources, and the transient time spent in the kernel is very short anyway, so, a preemptive kernel may not really be worth the effort. Thus, in hindsight, this idea should be reconsidered.
- Multiprocessor: With the advent of multicore-CPUs, embedded multiprocessor systems are becoming feasible. The PikeOS microkernel was not designed for multiprocessors, so this must be remedied. Besides being a necessary step to support future processor generations, this also opens up the interesting new prospect of finally solving the conflict between time-driven and event-driven threads by binding them to separate processor cores.
- Certification: An effort to certify an appliance using PikeOS to DO-178B ([20]) is currently underway.

References

[1] ARINC. Avionics Application Software Standard Interface. Technical Report ARINC Specification 653, Aeronautical Radio, Inc., 1997.

[2] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization, 2003.

[3] G. Bollella. *Slotted priorities: supporting real-time computing within general-purpose operating systems*. PhD thesis, University of North Carolina at Chapel Hill, 1997. Advisor: Kevin Jeffay.

[4] J. Bonwick and Sun Microsystems. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1997.

[5] D. Dodge, A. Dank, S. Marineau-Mes, P. van der Veen, C. Burgess, T. Fletcher, and B. Stecher. Process scheduler employing adaptive partitioning of process threads. Canadian patent application CA000002538503A1, March 2006.

[6] D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel data - first class citizens of the system. Technical Report PA005847, National ICT Australia and University of New South Wales, December 2005.

[7] G. Fohler. Flexible Reliable Timing - Real-Time vs. Reliability. In *Keynote Address, 10th European Workshop on Dependable Computing*, 1999.

[8] A. Haeberlen. User-level Management of Kernel Memory. In Proc. of the 8th Asia-Pacific Computer Systems Architecture Conference, Aizu-Wakamatsu City, Japan, September 2003.

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Modular System Programming in MINIX 3. *j-LOGIN*, 31(2):19–28, April 2006.

[10] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Diploma Thesis, TU Dresden, August 1996.

[11] M. Hohmuth. The Fiasco kernel: Requirements definition, 1998.

[12] R. Kaiser. Scheduling Virtual Machines in Real-time Embedded Systems. In S. A. Brandt, editor, *OSPERT 2006 Workshop on Operating Systems for Embedded Real-Time applications*, pages 7–15, July 2006.

[13] R. Kaiser and R. Fuchsen. Verfahren zur Verteilung von Rechenzeit in einem Rechnersystem. German patent DE102004054571A1, November 2004.

[14] R. Kaiser, S. Wagner, and A. Zuepke. Safe and Cooperative Coexistence of a SoftPLC and Linux. 8th Real-Time Linux Workshop, Lanzhou, China, September 2006.

[15] J. Liedtke. Clans & chiefs. In *Architektur von Rechensystemen, 12. GI/ITG-Fachtagung*, pages 294–305, London, UK, 1992. Springer-Verlag.

[16] J. Liedtke. On μ-Kernel Construction. In *SOSP*, pages 237–250, 1995.

[17] J. Liedtke. L4 Reference Manual - 486, Pentium, Pentium Pro, 1996.

[18] J. Liedtke, N. Islam, and T. Jaeger. Preventing Denial-of-Service Attacks on a μ-Kernel for WebOSes, 1999.

[19] NICTA – National ICT Australia. Embedded - ERTOS. Online: http://ertos.nicta.com.au/research/l4/embedded.pml, 2006.

[20] RTCA. Software Considerations in Airborne Systems and Equipment Certification. Guideline DO-178A, Radio Technical Commission for Aeronautics, One McPherson Square, 1425 K Street N.W., Suite 500, Washington DC 20005, USA, March 1985.

[21] S. Ruocco. Real-Time Programming and L4 Microkernels. National ICT Australia, 2006.

[22] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.

[23] H. Tuch, G. Klein, and G. Heiser. Os verification — now! In M. Seltzer, editor, *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.